DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

RELATIONAL PROGRAMMING:
DESIGN AND IMPLEMENTATION OF
A PROTOTYPE INTERPRETER

by

John R. Brown
and
Stephen G. Mitton

June 1985

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution is unlimited

T222816

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Relational Programming: Design and Implementation of a Prototype Interpreter	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): John R. Brown and Stephen G. Mitten	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	12. REPORT DATE: June 1985
	13. NUMBER OF PAGES: 227
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

relational programming

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Relational programming is a methodology which combines the advantages of functional programming with the relatively simple laws which govern relations. The goal is to give the programmer an environment which allows a higher level of programming abstraction than currently exists, an easier approach to proving programs correct, and a language which can support new parallel architectures. In this report, (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

the design and implementation of a prototype interactive interpreter for a relational programming language is presented. The reasoning behind the decision to use LISP as the implementation language is presented followed by an in depth discussion of the design inssues involved and the implementation decisions made. How to use the interpreter and future research topics are discussed. Also several appendices are provided which include the grammar, the relational operators implemented, and the documented LISP code.

Approved for public release; distribution unlimited.

Relational Programming: Design and Implementation of a Prototype Interpreter

by

John R. Brown
Captain, United States Army
B.S., Rochester Institute of Technology, 1975

and

Stephen G. Mitton
Lieutenant Commander, United States Navy
B.S., Rhode Island College, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

ABSTRACT

Relational programming is a methodology which combines the advantages of funtional programming with the relatively simple laws which govern relations. The goal is to give the programmer an environment which allows a higher level of programming abstraction than currently exists, an easier approach to proving programs correct, and a language which can support new parallel architectures. In this report, the design and implementation of a prototype interactive interpreter for a relational programming language is presented. The reasoning behind the decision to use LISP as the implementation language is presented followed by an in depth discussion of the design issues involved and the implementation decisions made. How to use the interpreter and future research topics are discussed. Also several appendices are provided which include the grammar, the relational operators implemented, and the documented LISP code.

TABLE OF CONTENTS

I. INTRODUCTION 8

II. BACKGROUND 11

III. WHY LISP?. 17

A. THE INTERLISP-10 PROGRAMMING ENVIRONMENT . . . 17

B. SCANNER AND PARSER IMPLEMENTATION SIMPLIFIED . 19

C. LISP PROVIDES BUILT-IN MEMORY MANAGEMENT . . . 21

D. RPL DATA STRUCTURES SIMPLIFIED 22

IV. RPL GRAMMAR AND SYNTAX 24

A. INTRODUCTION 24

B. DISCUSSION ABOUT THE ORIGINAL GRAMMAR 25

C. GRAMMAR MODIFICATIONS DUE TO LISP 27

D. GRAMMAR MODIFICATIONS DUE TO DESIGN AND IMPLEMENTATION 28

E. INFIX VS PREFIX OPERATORS 35

V. INTERPRETER DESIGN AND DEVELOPMENT 37

A. RPL PRIMITIVES 37

1. Individuals. 37

2. Sets 38

3. Relations. 38

a. The Evolution of 'Erel'. 39

b. The Sequence Loses Significance . . . 43

4. Functions. 46

B. RPL ENVIRONMENT 47

C. PARSING RPL 50

D. EVALUATING RPL EXPRESSIONS 53

1. The EV Function. 54

2. Evaluate Operands - PREFIXOP and INFIXOP.. 58

3. Binding Formals and Evaluation - RPAPPLY.. 59

4. Built-in Functions - Handled By BIF-APPLY. 60

5. Special Syntax - EV_SPECIAL_CASES. 62

F. EXTENSIONAL MECHANISM 66

G. ERROR DIAGNOSTICS AND RECOVERY 69

1. Error Recovery 69

2. RPL Diagnostics. 71

3. Errors Can Be Easily Built In. 72

H. INPUT / OUTPUT 74

1. Console Input/Output 74

2. File Input/Output. 77

VI. USING THE RPL INTERPRETER. 79

A. INTRODUCTION 79

B. GETTING STARTED 79

C. SESSION TERMINATION 82

D. EXECUTING COMMANDS 84

E. DATA DEFINITIONS 86

1. Introduction 86

2. Sets 87

3. Relations 87

4. Sequences. 88

5. Lists. 90

6. Ranges 90
F. FUNCTION DEFINITIONS 92
G. INPUT/OUTPUT 94
1. Screen Input/Output. 94
2. File Input/Output. 95
3. Debugging. 95
H. RELATIONAL OPERATORS 97
I. BEWARE THE KEYSTROKE 98
1. Introduction. 98
2. The Control-D (^D) and Control-C (^C). . . 99
3. The Backspace Key. 99
4. The Control-Z (^Z) 100
VII. RECOMMENDATIONS AND CONCLUSIONS. 102
A. USING BREAKDOWN 106
APPENDIX A - ORIGINAL RPL GRAMMAR 114
APPENDIX B - IMPLEMENTED RPL GRAMMAR 116
APPENDIX C - RPL OPERATORS 119
APPENDIX D - INDEX TO RPL OPERATORS 161
APPENDIX E - RPL INPUT FORM SUMMARY. 165
APPENDIX F - RPL CODE AND DOCUMENTATION. 169
APPENDIX G - EXAMPLES OF RPL PROGRAMMING 220
LIST OF REFERENCES 226
INITIAL DISTRIBUTION LIST 227

I. INTRODUCTION

Relational programming is a programming style which uses the relation as the basic structure for all programming. This innovative methodology may be a sound approach to meeting the future needs of the computer science community. Because entire relations are manipulated instead of individual data elements, relational programming may serve as the basis for an efficient, modern machine architecture which will overcome the limitations and low level word-at-a-time processing of the von Neumann type computers.

A relational programming language is a higher level language than conventional languages such as Fortran, Pascal, and Algol. These languages are sequential in nature and involve the programmer in many low level programming decisions such as keeping track of counters or indices to array structures. This means that the programmer must worry about how to manipulate individual members of an array to achieve the desired result instead of being able to deal with the array structure as a whole. Relational programming frees the programmer from these types of decisions, allowing him to work at a higher level of abstraction, concentrating more on WHAT the program must do, but not details of HOW it will be done. Relational programming can do this because data and programs are not treated differently. Data and

programs are equivalent since they are based upon a common structure, the relation.

The relation is a reasonable and feasible basis for a programming language because a well developed theory of relations exists and the laws which govern relations are relatively simple. A similar approach to relational programming, which has been an active area of research, is functional programming. Backus described in his Turing Award paper [Ref. 1] a functional language, FP, and its advantages in meeting future programming needs. As MacLennan [Ref. 2] has stated relational programming subsumes functional programming because every function is a relation. Therefore everything that can be done in a functional language can be done in a relational programming language. MacLennan has described the advantages of relational programming and demonstrated its potential as a powerful high level language. These advantages are summarized below:

1. Relational programming supports abstract higher level programming.

2. Relational programming deals with a single kind of entity, the relation, and uses it for all purposes.

3. Relational programming more directly supports non-linear data structures such as trees and graphs.

4. Programs can be algebraically derived and manipulated.

5. Relational programming can more easily support utilization of associative and active memories.

This research will serve as a mechanism to demonstrate the practicality and feasibility of a relational programming language as described by MacLennan [Ref. 2]. Therefore, familiarity with his report is necessary to better understand the further development of his work presented here.

This report will describe the development and design of a prototype interactive interpreter for a relational programming language. It will also demonstrate that such an interpreter is implementable on a current machine architecture, although it would probably be more suitable to a newer type of architecture.

This research and its product, an interactive Relational Programming Language (RPL) interpreter, will serve as a kernel and impetus for follow-on work with relational programming concepts. It is hoped that the issues and decisions made in this implementation will provide the answers to some of the basic questions, and identify some critical areas for future research.

II. BACKGROUND

The von Neumann model of computation has been dominant for the last 30 years and has remained largely unchanged even though significant advances in both software and hardware technology have taken place. Applications continue to become more complex and sophisticated, requiring increasingly more powerful computer systems. To date, extensions of conventional software systems have seemed to meet the demands. However, it has become quite clear that an alternative to the von Neumann computer organization is needed.

Programming languages were originally designed for and have supported the von Neumann machine architecture. But, as technology has advanced, the von Neumann sequential word-at-a-time bottleneck has become painfully apparent. Real world applications are not sequential in nature and the conversion of concurrent processes to operate sequentially affects efficiency and speed of computation.

Hardware research has acknowledged that a fundamental limit exists on the performance increases which can be derived from advances in technology alone. VLSI technology seems to be naturally suited to new types of parallel architectures, and programming language design is following suit with the development of higher level programming

languages which are more powerful, abstract and easier to prove correct. The increasing complexity of real world applications is dictating a need for higher levels of abstraction so that the programmer can concentrate on the overall solution without becoming bogged down in the details. Relational programming is one possible solution to this problem.

Relational programming is based upon the use of a relational calculus which can model almost any data structure. Therefore, the high level relational operators can also be used to manipulate entire data structures. MacLennan has presented and discussed the basis for a relational programming language in references 2 and 3. The operators he describes are based on naive set theory and operate on three basic objects: individuals, binary relations and sets. Individuals are the indivisible data values which can be used to compute. A binary relation is some property which relates one object to another. For example, the less than (<) relation relates all pairs of values, x and y, for which x is less than y. Therefore the pair (3,4) is a member of the '<' relation. The '+' relation can be denoted (x,y)+z, which means that it takes a pair (x,y) and relates it to its sum z. In general, a relation can be represented by the notation, xRy where x and y may represent any objects.

A set is any grouping of individuals, binary relations and/or other sets. Thus there is no restriction on what sets or relations can be members of other sets and relations.

With these basic objects, MacLennan develops and describes the operators which he feels would be useful to the relational programmer, and demonstrates the potential advantages of a programming language based upon a relational calculus. He shows that relational operators can be algebraically manipulated to derive other, more complex operators. This ability supports the premise that relational programs would be easier to prove correct. It also demonstrates that programs can operate on other programs to yield relatively straight-forward solutions to complex problems. High level abstraction is thus supported, allowing the programmer to be more productive and able to conceptually manage larger and more unusual applications.

An important point made by MacLennan is the need to separate intensional and extensional operators. Relations, functions and sets can have both a finite (extensional) or an infinite (intensional) representation. Many operators or combinations of operators are implementable in either representation. This complicates the programmer's life because he must remember the underlying restrictions involved when he wishes to use an operator which falls into one or the other category.

In order to prevent confusion caused by double duty operators, MacLennan made a decision to separate the operators into disjoint classes, those which are used on finite sets and relations, and those which operate on the computable functions which represent infinite sets and relations. For example the application operator can both be used for applying a function to its argument and for looking up an item in a table (a finite relation). The first case is represented f@x, which applies the computable function f to the argument x. The more common mathematical notation is f(x). The second case, which is denoted by t ↓ x, and read as 't select x', applies the finite table t to x. This simply means lookup x in table t and return the first item related to x. Thus, if t = (1:2, 2:3, 3:4, 4:5) and x = 2, t ↓ x would return '3'. The ':' operator used above is just a pair making operation which says the x:y is a pair (x,y) that is a member of the relation R, hence xRy.

The operators were further subdivided by MacLennan into a primitive class and non-primitive class. Operations were considered to be primitive if they could not simply be defined in terms of other operations. 13 primitive extensional operators and 15 primitive intensional operators were proposed by MacLennan. These primitive operations were supplemented by 55 non-primitive extensional operators, 10 non-primitive intensional operators and 13 miscellaneous operations which were defined in terms of the primitive

operators. MacLennan felt that these non-primitive operations should be built-in to any relational programming language implementation. Because the work done in this study resulted in modifications to some of the operators proposed by MacLennan, a discussion of the operators will be presented in later chapters and in detail in Appendix C.

Since a computer's memory is finite, representation of large extensional sets and relations is of major concern. To this end, Suha Futaci [Ref. 4] extended MacLennan's research by analyzing the complexity of the algorithms associated with several different extensional representations.

Finally, the purpose of the prototype interpreter developed in this research is to further advance the study of a relational calculus as a programming methodology. The interpreter will provide a tool to evaluate the relational operations and provide tangible input for the selection of optimal set of combinators and relational operators. To achieve this several unique linguistic issues made the implementation of this prototype particularly interesting:

1. RPL supported a syntax which allowed infix operators to be used in prefix format if desired. The expressions (x + y) and [+]<x,y> have the same semantics, therefore the parser had to be designed so that both expressions were ultimately evaluated by the same function. The utility one can gain by this convention is illustrated in Example 1 of Appendix G.

2. Many operators can be defined in RPL which require the creation of huge sets or relations to be generated as an intermediate form. This is generally what may

happen before the application of a filtering operator, in which the final result requires a fraction of the storage needed by the intermediate form. This is illustrated by the development of the 'xi' operator, see Example 2 Appendix G. A mechanism to allocate storage and perform garbage collection is imperative for RPL. Such a mechanism was provided by LISP's built-in storage management system. Having this feature available in LISP was a major consideration for its use as an implementation language.

3. The original grammar shown in Appendix A was not deterministic and had several productions defined with left recursion. It also contained several meta symbols that had special meaning to LISP (these included '(', ')', '[', ']', and '.'). These issues resulted in the transformation of the grammar to the one shown in Appendix B.

4. Twelve of the fourteen alternatives to the production 'primary' shown in Appendix B are tagged LISP lists. This syntax provides a deterministic way of parsing these entities and alleviates the problem presented with the LISP metasymbols contained in the original grammar. Having tagged lists for these structures in RPL led to a type checking mechanism where most of the RPL primitives are implemented with a unique identification tag.

Chapters III through V will further examine these issues and outine the overall design of this prototype. Chapter VI explains how to use the interpreter and provides several sample terminal sessions for illustration. Chapter VII demonstrates the use of LISP performance analysis features and suggests a direction for follow on research in RPL.

III. WHY LISP?

There were four primary considerations for using LISP as an implementation language for the RPL interpreter: the availability of the Interlisp-10 programming environment, the ability to simplify scanning and parsing by adopting a LISP-like syntax, the ability to use LISP's built in memory management and garbage collection system, and finally, the ability to simplify several complex data structures by using built in LISP structures.

These advantages far outweigh the sometimes awkward LISP-like syntax, and some of the LISP specific input/output problems that surfaced as the prototype was developed. A discussion of the all RPL input/output, including the problems encountered, is found in Chapter V.

A. THE INTERLISP-10 PROGRAMMING ENVIRONMENT

The Interlisp-10 system provides a rich programming environment. The tools it provides to enhance code development include an integrated structure editor, a compiler and an excellent set of debugging facilities. These tools operate within a framework which does more than just process one command and wait for the next. Three additional resident features of Interlisp that are always

present to enhance program development also influenced the choice of LISP as an implementation language.

The 'Do What I Mean' (DWIM) feature of Interlisp, is invoked any time the system detects an error. DWIM attempts to correct common programming errors by trying to logically predict what the programmer had intended. The ability of the DWIM feature to correct spelling and typographical errors is a definite time saver.

Another resident feature of the Interlisp environment is the Programmer's Assistant (PA). This feature basically maintains a history list of all commands entered by the programmer. Using various PA commands the programmer can REDO a sequence of operations, or use UNDO to cancel previous operations, or replace one variable name with another with the USE command.

Two particular features available in the Interlisp environment, Masterscope and Breakdown, are especially useful to future reasearch. Breakdown is an excellent tool for conducting performance analysis, allowing the programmer to probe the system to collect information such as, the number of calls and amount of cpu time required by a particular function. The programmer can even find out how many times a function executes another function (sometimes the number of calls on the LISP CONS function is a good performance indicator in LISP systems).

Masterscope is a remarkable feature of the Interlisp environment which creates a database from analyzing a program. Using this database, the programmer can interrogate the system to find out information, such as where each function is called and where variables are bound or referenced, or edit a function any where a particular variable is used. This feature is particularly desirable in a prototype such as this since follow on research will have a facility to predict the effect of changes as program revisions are proposed and implemented.

B. SCANNER AND PARSER IMPLEMENTATION SIMPLIFIED

Since LISP views everything in terms of its primitives, atoms and lists, the tokenization function normally provided by a character-at-a-time scanner was significantly simplified, although the grammar had to be modified slightly to adopt a more LISP-like syntax. By requiring all expressions to be enclosed within a set of parentheses, parsing an expression becomes a simple matter of determining the length of an expression. The LENGTH function is built into LISP. For example an infix expression written as (x + y) is recognized by the length 3, while the prefix expression (not p) is distinguished by its length of 2. Notice the requirement for spaces between the operand and operator. Spaces and parentheses are the only delimiters used in RPL's LISP-like syntax. Although this syntax became

necessary as a result of implementation issues, it served the main objective of this prototype, to develope a tool to further advance the study of the use of a relational calculus as a programming language.

The ability to readily identify infix and prefix expressions provided a logical basis for the overall design of the parsing function.

By representing all RPL expressions as LISP lists, extracting the operands and operator of a given expression can be accomplished easily by using the LISP CAR and CDR functions. These functions each take a non-empty (non-null) list as its argument. The CAR function returns the first element of a list, whereas the CDR function returns a list containing all elements of a list except the first element. Therefore, the CAR function is used to extract the operator of a prefix expression, and the operand is obtained by first using the CDR function on the expression, followed by the CAR function. For example, the expression (not p) can be parsed into its operator and operand as follows:

```
operator <= (CAR '(not p)) = not
operand  <=  (CAR (CDR '(not p) = p
```

Note that LISP evaluates nested functions from inside out. This means that to obtain the operand, the function (CDR '(not p)) is evaluated first, which returns the list (p). This result is then the argument to the CAR function, which extracts the p from (p). Since LISP programming requires

many instances where successive CAR and CDR combinations are required, a shorthand notation simplifies the operand extracting code to the following:

operand <= (CADR '(not p)) = p

where the 'A' of the CADR function comes from the CAR function, and the 'D' from the CDR function.

Therefore, simple length checks on expressions direct the parse into two logical subsets. Once this is accomplished the operator and operands are readily accessible through a sequence of CAR and CDR function calls. This simplicity made LISP particularly attractive as an implementation language.

C. LISP PROVIDES A BUILT-IN MEMORY MANAGEMENT SYSTEM

Using LISP as an implementation language also eliminated the need for coding a memory management and garbage collection system, since these features are already available in LISP. Issues such as variable storage requirements simply went away. The ability to let a proven system like Interlisp perform all the memory management provided a sound foundation on which the RPL system could implemented. This also eliminated a very error-prone area of coding that might have created significant delays in the development of this prototype.

D. RPL DATA STRUCTURES SIMPLIFIED

Many of the data structures needed by the RPL interpreter were readily available in LISP. Using built in LISP functions simplified and/or eliminated a considerable amount of code in the sets and symbol table data structures.

ALL of RPL's extensional operators operate on finite sets. LISP's implementation of sets is simple, the LISP list. Additionally, Interlisp-10 provides a complete assortment of set operations including union, intersection, set difference, cartesian product and both membership and subset boolean functions. Using these built in LISP functions as a foundation, all that was needed to implement many of the set operators in RPL was the addition of type checking to ensure the compatibility of the operands used with the built-in functions.

One of the main design decisions in the development of the RPL interpreter was the choice of the data structure to represent the symbol table. Several related design decisions had already decreased the complexity of the symbol table requirement. Variable storage requirements were no longer an issue, and a type checking tag was to be embedded within the variable's definition. All that was needed was a mechanism that could provide a binding between a variable name and its definition, along with a fast and efficient accesssing function to retrieve the definition of a variable given its name and scope. This requirement translated directly to the

LISP association list, or a-list. The RPL symbol table is referred to as the RPL environment (denoted globally as 'E') since it is the same structure used in MacLennan's development of a LISP interpreter written in LISP, [Ref. 5].

The a-list is nothing more than a list where each element is a list. The following is an example of an a-list:

E = ((x 1) (y 2) (z 3) (t set 1 2))

Each element of the a-list represents a name/definition pair. The name is the CAR of the a-list element, its definition is the CDR. In the example above the x, y and z are bound to 1, 2 and 3 respectively, while t is bound to (set 1 2).

The a-list structure in LISP can be efficiently scanned by the LISP SASSOC function. This function, given an a-list and a target, will return the a-list element (both target and its definition), if the target name is found, otherwise it returns NIL, indicating the target was not in the found.

The use of the a-list data structure to represent the RPL environment provided still another means to simplify the the overall coding requirements of the interpreter.

IV. RPL GRAMMAR AND SYNTAX

A. INTRODUCTION

One of the goals of relational programming is to develop a notation which is both readable and has the manipulative advantages of a two-dimensional algebraic notation. Such a notation would enhance the ability of relational programs to be more easily proved correct. Unfortunately, most printers do not incorporate the unique mathematical symbols that are necessary to support a notation of this type. However, there are software methods which enable some specialty printers to produce such symbols.

With such a notation in mind, MacLennan proposed the original grammar shown in Appendix A. This grammar was printed using the 'eqn' package of the Unix Operating System. This package is a text formatting tool which takes an English-like description of an equation and generates the mathematical symbols for that equation when it is printed. Thus the notation and operator names utilized by MacLennan have the eqn input format as a base. The utility of the eqn package is introduced in this version of the grammar, but its real value will be demonstrated later when the symbols selected for the operators are discussed.

MacLennan's grammar accurately presents the production rules necessary to produce legal relational programming

statements independent of implementation considerations. However, it is loaded with left recursion, which means a great deal of effort would have been required to transform it into a form from which a conventional parser could be generated. Fortunately, the decision to use LISP as an implementation language eliminated this concern, but did present other problems which required modifications to this generic grammar. In addition to basic changes required by the use of LISP itself, other modifications were found to be necessary as the RPL interpreter was designed, tested and exercised. The remainder of this chapter will discuss the evolution of the original grammar into its implemented form presented in Appendix B.

B. DISCUSSION ABOUT THE ORIGINAL GRAMMAR

At the highest level, the original grammar called for an interactive session which consisted of zero or more commands and the word 'done'. Commands could consist of a data definition, a prefix function definition, input from a file and output to the screen. In addition to the many built-in infix and prefix operators, several special constructs were available including iteration, superscription and conditionals. Finally, a variety of symbols represented different objects within the language.

The bracket symbols, '[' and ']', had two meanings as printed in Appendix A. In one sense their use meant that

the object(s) enclosed were optionally required. This meaning is still retained in the revised grammar. On the other hand the brackets also were terminals in the language which produced different relational structures depending upon what objects were enclosed by them. First, an infix operator enclosed in brackets, e.g. [+], transformed the '+' operator which took two numeric arguments, into a prefix operator which took one argument, a pair of numbers. Thus (x + y) became equivalent to [+](x,y) where x and y could be any number. Second, the brackets could be used to fix either the left or right arguments of an infix operator. Therefore, it was permissible to write [3+]x where [3+] is a specialized operator which adds '3' to any other single numeric argument such as x. Likewise, [+4] fixed the right argument to '4' and would add any numeric argument provided to '4'. Use of the brackets in any of the above manners created a functional which could be combined with other functionals to create whatever mechanisms were required to accomplish a particular task.

Parentheses were included to allow natural mathematical groupings of both expressions and their arguments. Thus expressions could be both RPL functionals or data. The angle brackets, '<' and '>', when used to enclose data represented a special sort of sequence which had a termination symbol, much like a LISP list structure which ends in 'nil'. Finally, braces were used to enclose the elements of a set.

The use of these symbols presented a convenient method for manipulating functionals, but conflicted with the LISP syntax. The changes to the grammar that resulted because of this are discussed next.

C. GRAMMAR MODIFICATIONS DUE TO LISP

Unforturnately, parentheses and brackets have a different meaning in LISP.. In LISP parentheses are used to delimit a list structure. Brackets serve basically the same purpose, but the right bracket, known to some as the super bracket, closes off all left parentheses which do not have a matching right parenthesis. For those who are familiar with LISP, this feature is both good and bad! Some say LISP stands for 'Lots of Idiotic Stupid Parentheses' which summarizes the frustrations encountered with parenthesis bookkeeping.

This conflict of symbols required that an alternative syntax be developed to conform with the LISP list structure and still maintain the semantics of the RPL language. To distinguish between structures, it was decided to use keywords as the first element of the list which represented them. These input formats are then transformed into the internal structures required by the interpreter. Another problem was the use of a pair of dots or periods to indicate a range of values. For example, in the original grammar the range (6..8) was equivalent to sequence (6,7,8). Use of the

character '.' in RPL created a symbolic conflict in LISP. Dots in LISP are treated as special connectors which form a structure called a dotted pair. Since LISP does not normally treat dots as regular characters, anywhere a pair of dots was required in the original grammar, the word 'to' was substituted in the new grammar.

Although some of the symbols used in the original grammar did not pose a problem in LISP, they were abandoned for consistency. The resulting constructs are summarized by example in Table IV-1. Note that these formats are just LISP lists with their formal requirements for spaces between the objects in the list, be they numbers, words or any grouping of characters. Thus, a disadvantage of LISP is inherited by RPL, the importance of spaces and the correct placement of parentheses.

D. GRAMMAR MODIFICATIONS DUE TO DESIGN AND IMPLEMENTATION

Several productions were added to the grammar due to considerations and factors which surfaced during the implementation process. At the command level a decision was made early on to increase the flexibility for the RPL programmer by allowing him to define infix operators as well as prefix operators. The original grammar forced the programmer to define infix operators in a prefix format. That meant that his normal thinking about an infix operator had to be altered to fit the prefix form of a function which

Table IV-1 -- RPL Grammar Modifications
Required By Use Of LISP

RPL SYNTAX	
Original	Final
[+]	(op +)
[3+]	(lsec 3 +)
[+4]	(rsec + 4)
(1,2,3,4,5)	(seq 1 2 3 4 5)
(1..5)	(seqrange 1 to 5)
{1,2,3,4,5}	(set 1 2 3 4 5)
{1..5}	(setrange 1 to 5)
<1,2,3>	(list 1 2 3)
<1..5>	(listrange 1 to 5)
iter[p->f]	(iter p -> f)
[if p -> f;g]	(if p -> f ; g)

takes a single argument - in this case a list containing two arguments. Internally, all operators can be considered as a prefix, but most people have become accustomed to thinking about binary operators in the infix sense. For example, to add 2 and 3 in RPL it is natural to write '(display (2 + 3))'. But to define the infix operator 'plus' which would do the same thing, a user would have to enter '(plus (x y) == (x + y))'.

To alleviate this inconsistency, a production rule was added to allow the programmer to define the operator 'plus' in the more natural way and to use it the same as any other infix operator:

```
Definition =>     (x plus y == (x + y))
Example    =>     (display (2 plus 3))
```

The second major addition to the grammar was a similar construct to the LAMBDA expression in LISP. This construct provides the programmer with a great deal of flexibility and was incorporated into RPL as a 'func' expression to insure no confusion with the LISP equivalent. Like the LAMBDA expression in LISP, the func expression consists of the name of the function, a list of formal parameters, and the body of the function in terms of the formals. Thus, the RPL programmer can now define functions/relational operators in three ways, directly using the 'func' expression, as a prefix operator, or as an infix operator. For comparison, the three types of definitions for the 'plus' operator as

described on the previous page are shown below:

```
Direct:  (plus == (func (x y) (x + y)))

Prefix:  (plus (x y) == (x + y))

Infix:   (x plus y == (x + y))
```

From the examples above, it appears that parentheses are going to plague RPL just as they do LISP, but, as will be discussed in a later chapter, the Interlisp environment provides a mechanism which allows the outside parentheses to be dropped when inputting commands, and actually assists in keeping tract of correct placement of parentheses.

The next modification, which was deemed appropriate to make the programmer's life a little easier, dealt with the RPL command 'display'. At the command level this word had to be written to obtain output to the screen. It quickly became apparent that it was cumbersome to type 'display' in order to see every result of a computation. So, the alternative input forms of 'dis' and 'd' were added. Finally, even these forms were made optional, requiring the interpreter to detect automatically the programmer's intent.

As mentioned earlier in this chapter, the original grammar only permitted input from a file. The intent was to allow the user to create a series of RPL data structures outside of the RPL environment and to read them in as necessary during a session. It became apparent that there also was a need to save data created during a session. For example, a database in RPL is just a large set of records,

where each record is a relation between the field names and their associated values. It is desirable to be able to read the entire database structure from a file, update it in some fashion during a session, and rewrite it back to a file. To allow this, another production was added at the command level which permitted commands of the form:

file string == expression

Execution of a command of this type would place the value of the evaluated expression into a file with a filename given by the 'string' argument of the operator 'file'. For example, consider an existing database stored in a file called 'OldMaster' and an updating function, called 'Update', which when given a database as an argument would modify the value of a selected field in all records and return the updated database. With this new production it is then possible to execute the following command:

```
(file "NewMaster" == (Update (file "OldMaster")))
```

This command would read the 'OldMaster' file in, execute the 'Update' function with 'OldMaster' as its argument and then rewrite the updated database to the file 'NewMaster'.

The one problem with this construct is that it should not be used to store function definitions to a file. A function definition has associated with it an environment of definition. This environment consists of all previously defined functions, their environments, and any data definitions made up to the point of definition in the

session. Since the environment is nothing more than an association list which contains the bindings of all names to their values, this list can become extremely long in a short period of time. Internally, pointers are used to conserve space, but when printed, the entire environment chain is produced, which could result in many pages of information. As discussed in Chapter V this could cause a fatal problem or be a terrible inconvenience at the least. Another feature of the RPL system, which is discussed in more detail in Chapter V, allows function definitions created during a session to be saved for future use and thus avoids the problems which could be created with the file command in the output mode.

The function definition and its associated environment did lead to two other grammar modifications. First, the initial implementation of the 'display' command returned the evaluated form of the argument. Therefore, the result of executing such a command returned something totally different from what the user typed in and compounded the problem with environment length.

For example, say the user typed in the following data definition:

```
(x == (seq 1 2 3))
```

Later in the session he decides to remind himself of how x was defined. He types in the command (display x), but what is returned is not his definition, but the internal

representation of the sequence he defined:

(Erel (1 2) (2 3))

Likewise, if he had defined the function f as:

(f x == (x times x))

and then entered (display f), he would see:

closure x (x times x) ...

Internal representations will be discussed in detail in Chapter V. To an unfamiliar user, this would be quite confusing and so the DISPLAY function was modified to return the user definition as it was typed in.

After one becomes familiar with the RPL language it becomes desirable to sometimes see the evaluated internal representation of any particular name. This feature is especially helpful when trying to debug a command that didn't work. The 'val identifier' command was developed to handle this need and was extended to meet the need to see the overall session environment or the environment of any particular function.

Every function definition has its environment of definition attached when it is converted into its internal representation. In LISP, that means a simple pointer is added to the list which describes the function. When this definition is printed, however, that simple pointer is the beginning of a very long list of pointers which may represent atoms or other lists of atoms to be printed. Consequently, pages of information are printed to the

screen. When this same information was in evaluated form, the result was excessive and usually resulted in aborting the session. To prevent this surge of unwanted information, the DISPLAY function was modified to print only the first three elements of a function definition, its name, its formal paramenters, and its body. Unfortunately, this modification also eliminated the ability to ever look at any environment. So, the 'env' command level productions were created. They allow the user to look at the overall session environment or the environment of any designated function. These features will be discussed further in the input/output section of Chapter V.

E. INFIX VS PREFIX OPERATORS

At first view the myriad of operators shown in MacLennan's grammar seem overwhelming and confusing, but one must remember that many of the words and symbols chosen were based upon the Unix eqn input format. Due significantly to the way the RPL interpreter was developed, many of the prefix operators became more naturally suited to an infix format. Some operators were discarded as no longer relevant because of changes in the way argument lists were represented. Others were added because of a new found utility based upon the same change just mentioned. It is also here where the true utility of the eqn text formatting tool becomes apparent. The sheer quantity of operations,

due mostly to the goal of preventing overloaded operators, required a great deal of distinct symbols. The purpose and use of these operators are discussed in Appendix C, but their names, original input forms, final input forms and the eqn publication forms are shown in Appendix E. This appendix summarizes the final changes to the grammar, highlights the conversion of some prefix operators to infix, and also serves as a concise guide to the relational operators and their syntax. Finally, the current grammar as implemented by the RPL interpreter is shown at Appendix B and includes all the modifications discussed in this chapter.

V. INTERPRETER DESIGN AND DEVELOPMENT

Previous chapters have illustrated the rationale behind the choice of LISP as an implementation language and the resulting modifications that became necessary to adapt the the original RPL grammar. The purpose of this chapter is to focus on issues related to the implementation the of RPL primitives and the overall structure of the interpreter. In addition, since MacLennan's report [Ref. 2] illustrates how many RPL operators can be implemented by defining them in terms of a set primitive operators, the mechanism used to implement the extensible nature of RPL is also an issue that will be discussed.

A. RPL PRIMITIVES

RPL contains three fundamental elements, individuals, sets and relations. The function, which is merely a special case of a relation, was added to the list of primitives because it required a unique internal representation.

1. Individuals

The indivisible data element found within RPL is the individual. This data type is equivalent to LISP atomic values and is implemented accordingly. Numeric, string and boolean scalars common to all programming languages are available in RPL. Strings must be enclosed in quotation

marks to distinguish them from LISP literal atoms. Literal atoms in LISP are used to implement the boolean values, 'true' and 'false', and all identifiers.

2. Sets

The set in RPL is implemented as a LISP list containing the tag 'Eset' as its first element. The tag 'Eset' is used both to distinguish the internal set representation in evaluated form from its input format and as a type checking device. For example, the set having the internal representation (Eset 1 2 3) may have been input as (set 1 2 3) or (set a b c), where a-c have appropriate internal bindings.

3. Relations

The finite relation, being a special kind of set, has an internal representation that closely resembles the set. The relation requires a special type of LISP list, called an association list or a-list. This particular data structure was chosen to implement the relation for two reasons. First, the mathematical notation for a relation closely resembles an a-list. For example, the mathematical relation

{ (1,2) (2,3) (3,4) (4,5) }

is represented in RPL as the following tagged a-list:

(Eset (1 2) (2 3) (3 4) (4 5)).

Second, the desire to use relations as tables, suggests the choice of a data structure that can be searched quickly and

efficiently. The LISP SASSOC function provides this capability when called with an a-list as its argument.

Since many built-in RPL operators are designed to operate on relations, to perform the fast recognition necessary for type checking, the 'Erel' tag was used in place of the 'Eset' tag. This efficiency was not free. The cost of distinguishing relations as a special type of set was paid for by the increased complexity in set operations and the coding necessary for coercion functions.

a. The Evolution of 'Erel'

During the earlier stages of development, after the decision to have the 'Erel' tag to distinguish relations, it seemed logical to extend this principle to special kinds of relations, namely sequences and arrays. There were many operators within RPL designed to operate on these kinds of relations, therefore, for the same rationale behind having the 'Erel' tag, the 'Eseq' and 'Elist' tags were adopted.

The language incorporated two input formats as convenient ways to enter mathematical sequences and arrays. The familiar mathematical notation for the two entities was reflected in the original grammar. The sequence was shown in the original grammar as (2, 4, 6, 8), whereas the array (n-tuple), was represented as < 2, 4, 6, 8 >. Both of these are mathematically relations:

(2, 4, 6, 8) <=> { (2,4) (4,6) (6,8) }

< 2, 4, 6, 8 > <=> { (1,2) (2,4) (3,6) (4,8) }.

This was modified to the following LISP-like syntax:

(2, 4, 6, 8) => (seq 2 4 6 8),

< 2, 4, 6, 8 > => (list 2 4 6 8).

For completeness, an input syntax was adopted to permit relations to be entered through the use of the tag 'rel', in place of the 'set' tag, and the use of the RPL pair making operator, ':'. The input format

(rel (1 : 2) (2 : 3)),

was represented internally in RPL as the relation

(Erel (1 2) (2 3)).

Although the decision to have different tags to distinguish each special kind of set made type checking very fast and efficient, having numerous internal forms that are mathematically equivalent was a problem not easily solved. Consider the relations r, s and l bound as follows:

r <= (Erel (1 2) (2 3))

s <= (Eseq (1 2) (2 3))

l <= (Elist (1 2) (2 3)).

Any operation applied to any of these relations should yield the same results. Additionally, an equality test comparing any two of them should return 'true'. This situation becomes even more muddled if the following binding is made:

s' <= (Eset (1 2) (2 3)).

Now there are four variables, bound to four physically

different representations, which must be evaluated as equivalent structures. This is like trying to do computation with numbers given four different number systems.

The problems created by having four objects with the same meaning was not solvable without a considerable amount of coding. A coercion function for every possible representation was required. The global variable 'ESETS', a list of tags considered legal for set operations, had to be established. Precedence rules had to be implemented to determine what tag to affix to the result of a set operation. The equality check had to be designed to focus on tagless lists. All this additional effort hardly seemed cost effective for a prototype, especially when the algorithm for the coercion function to create a sequence was considered. Coding to ensure a set is a fully connected irreflexive bijection (definition of a sequence used in by MacLennan [Ref. 2: p. 22]) is not trivial task.

It was time to re-examine the efficiency gained in the type checking mechanism by having tags distinguish various kinds of sets, versus the increased coding complexity necessary to ensure the semmantics is not altered in this new syntax. This involved screening MacLennan's report [Ref. 2] to classify operators based on their operands and their output. It was observed that when a prefix operator required two arguments, a two element sequence was used. For example, the function defined as

```
sum == (x + y)
```

was used with the syntax, sum(2,4). Further analysis found cases where the use of the sequence was inconsistent with its formal definition. This discovery led to the RPL list, depicted as <x,y> in the original grammar, replacing the seguence as the form for arguments to functions like 'sum'. This shift from sequences to lists will be discussed in more detail in the following subsection.

The significance of the shift from sequences to lists as functional arguments was that the sequence and its operators were now considerably less important to the RPL programmer. This, along with the coding complexity described earlier, resulted in the decision to abandon the 'Eseq' tag. Additionally, knowing a set is a relation makes it is easy to verify if the relation is a RPL list. This resulted in the elimination of 'Elist' tag also. By eliminating these two tags a viable compromise had been made.

The special input formats discussed previously were kept in the language for user convenience, with the tag 'Erel' being appended internally, vice 'Eseq' or 'Elist', to the a-list that made up the relation. Sequence operators were still provided but error checking was limited to verifying that operands are relations. This put the responsibility on the programmer to ensure proper arguments were used.

The end result of the trade-off analysis, weighing the issues of type checking efficiency verses code complexity, brought to light the detail and depth of planning required to design an effective software system. Language features are not free, and simple solutions to one problem may well create a snowball effect in complexity in other areas. Unfortunately, sometimes this is not obvious without modeling the implementation.

b. The Sequence Loses Significance in RPL

The sequence is used by MacLennan [Ref. 2] to represent an argument to multi-parameter prefix operators and functions. Many applications used the sequence operators, alpha and omega, to extract the individual operands from the two element sequences. In the sequence (x,y), alpha and omega were used to extract x and y respectively. These operators can only be used on a pure sequence. Graphically sequences can be represented as being a fully connected structure, with no cycles, and all arrows pointing in one direction (see Chapter VI).

In addition, the DELTA function was introduced to create a mechanism that could duplicate an argument for function application. For example, the squaring function would be defined as follows:

sqr == [times] o DELTA.

The DELTA function duplicates any argument returning the sequence shown

DELTA n => (n , n).

Therefore sqr 4 can be written

sqr 4 => [times] (4 , 4).

This looks perfectly reasonable, except that (4 , 4) is not a sequence. By definition a sequence is irreflexive.

The problems created by the irreflexive property of the sequence are discussed in MacLennan's research [Ref. 2: p. 22] in considerable detail. He also suggests an alternative definition to the sequence, but the structure used as the argument to functions remained sequences throughout.

The failure of the sequence as an argument to functions became obvious as many of the extensionally defined operators were implemented. In many instances definitions used the alpha and omega operators on their arguments. These functions would not work for inputs of the form (n, n). Functions that were defined in terms of DELTA, alpha and/or omega would not work on any input, since the operations

alpha o DELTA

and

omega o DELTA

are undefined.

The RPL list, which had notational similarities to the sequence, <x,y> verses (x,y), was a logical replacement to the sequence as the argument to functions. The list construct <n, n>, which is just an

array, was well defined, filling the void not covered by the sequence. The 'sel' operator, when used with RPL list, provided a means to extract each operand to a function, similar to the alpha and omega used previously on the sequence. The operators [sel 1] and [sel 2] extract the x and y components from the list <x,y>. DELTA had to be redefined to return the list <n,n>. The !! operator, which was defined as

f !! g => (f(x) , g(x))

was also refined to

f !! g => < f(x) , g(x) >.

Essentially, definitions where (x,y) appeared in the original report were replaced by <x,y>, and alpha and omega were replaced by either [sel 1] and [sel 2] , respectively.

The unsuitability of the sequence as a argument to a function has in no way diminished the power of RPL. The list structure is just as easy to manipulate algebraically, and is more versatile in many respects. For example with the use of the 'func' a programmer can define functions of the form

add3 == (func (x y z) ((x + y) + z)).

This can be used for any number of variables. A flexibility not possible with sequences. From a system development aspect, it is far easier to perform error checking on lists. If anything the shift from sequences to lists made RPL system development and programming tasks simpler.

4. Functions

Since RPL is extensible, both user defined functions and system functions that are defined in terms of a kernel of primitive functions have the same internal representation. This representation consists of four elements, the keyword 'closure', the formal parameters, a function body and an environment pointer.

The keyword 'closure' is adopted from its use by MacLennan [Ref. 5: pp. 436-437]. He defined a closure as having two elements, which can be used to implement static scoping, an instruction part (ip) and an environment part (ep). The ip is a pointer to the part of the code which defines the function, and the ep is a pointer to the context of a given function, which is all the names visible to that function. For RPL purposes the keyword 'closure' is merely a type checking tag like 'Eset' and 'Erel'. However, the basic structure used by MacLennan to implement static scoping in his model LISP interpreter was also used in RPL. Figure V-1 shows the parallel between MacLennan's model and RPL.

The formal parameters and the body of the function correspond to the ip used by MacLennan. Formal parameters are represented in LISP as either a literal atom or a list of literal atoms. The body of the function is a LISP list which is syntactically a RPL expression. The expression is

defined in terms of the formal parameters along with names defined in function's environment of definition.

The environment pointer is a snapshot of the RPL system environment at the time a function is defined. More precisely, this pointer corresponds to the RPL system environment pointer when the function was defined (this takes advantage of the way LISP implements the list internally). In view of this, all names defined by the RPL programmer during a session and all RPL built-in functions are within a function's environment of definition.

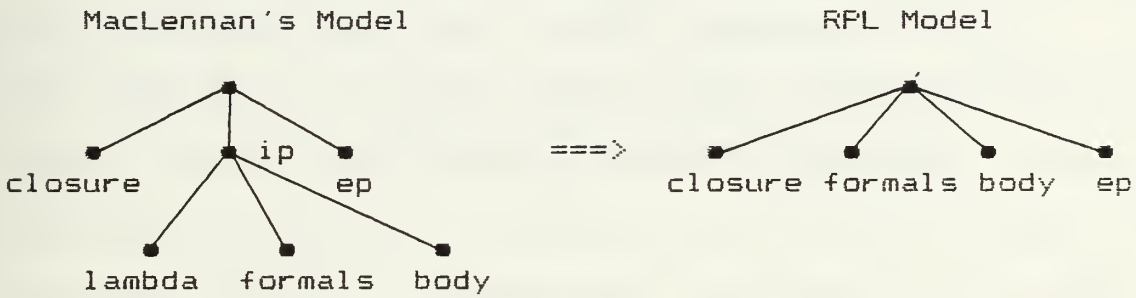


Figure V-1 -- Similarity between Models

Section D, which illustrates the process of evaluating funcions will elaborate on how RPL system binds formal parameters to their actuals.

B. RPL ENVIRONMENT

As discussed in Chapter III two of the main advantages for using LISP as an implementation language were the ability to use built-in LISP data structures and LISP's

memory management system. By embedding tags as part of the internal definition of sets, relations and functions, and using the LISP boolean functions, NUMBERP and STRINGP, on individuals (non-lists), the type checking mechanism was easily established. Therefore, many of the attributes normally stored in the symbol table of conventional language systems were eliminated. Combining this with the static scoping mechanism discussed in the previous section reduced the RPL symbol table requirements to a mechanism that would bind each name with a pointer to its internal definition and provide a fast means of accessing that definition.

LISP implements the list very efficiently by using pointers to cells in memory. Since every list can be broken into two components, its CAR and CDR, the list was a simple but logical choice of a structure to be used to associate a name with its definition. The name and its definition form a pair corresponding to the CAR and CDR of a list.

A list construction function, appropriately called CONS, is available in LISP. CONS takes two arguments, the first argument is the CAR of the list, the second argument is the CDR. Using this function a binding can be made between a name and its definition. This is illustrated in Figure V-2.

The most primitive of LISP lists is called a dotted pair. Like any other list, dotted pairs have a CAR and a CDR. Dotted pairs get their distinction from the dot that

sometimes separates the CAR and CDR when displayed. The CONS function usually adds its first argument to the beginning of a list, which is the second argument. Most LISP lists end with a NIL marker, thus (CONS 1 NIL) is the list (1). However, list without a NIL marker occurs when the second argument to the CONS function is an atom (and not NIL). Since the second argument has no NIL marker, the list created by CONS in this instance has no NIL marker either and it looks a little strange when printed. LISP prints its lists by following the pointers of each element. A dot '.' is printed preceding the last element if there is no NIL marker associated with it. This is why a dot is shown in the illustration (f . def). In many LISP implementations the dot '.' is the same operation as the CONS function.

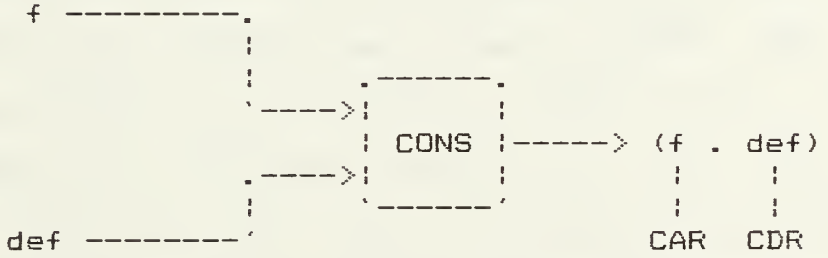


Figure V-2 -- Typical Binding

Having each name associated with its definition by using the CONS function is not a novel idea to a LISP programmer. A list of these pairs is called an association list or a-list in LISP. To search these constructs rapidly

LISP provides the SASSOC function. This function, when called with a target and an a-list, will compare the target to the CAR (or name pointer) of each element of the a-list. If the target is found the entire element is returned. Taking the CDR of this result provides the definition. This process is encapsulated by the RPL system function LOOKUP. The simplicity and efficiency of this data structure makes it an excellent mechanism to implement the RPL environment, especially in a prototype.

Although efficiency issues of RPL will be topics for future research, the design of the RPL environment using the the a-list owes its efficiency to its LISP implementation. By taking advantage of the characteristics of the LISP list and its most basic list constructor to bind names and definitions it was hoped that efficiency could be inherited from LISP. Pointers used in many PASCAL like languages are often hard to use and error-prone. LISP provides the efficiency of using pointers without the programmer having any conscious awareness of their implementation. This level of abstraction simplified the programming task considerably.

C. PARSING RPL

In most languages user input is first analyzed by a scanner. However, by using LISP as an implementaion language and making some minor modifications to the grammar to adopt a LISP-like syntax, the functionality of the scanner was

eliminated. The RPL command line is simply a LISP list. Using various LISP funcions to examine the syntax of this list, the semmantics of the command is extracted.

Parsing the grammar shown in Appendix B can be accomplished by dividing the parse into two stages. The basic input to the interpreter is the RPL command line. Determining which of the nine different commands is being used is the first stage of the parsing task. Five of the commands require the evaluation of a RPL expression. Parsing the expression is the second parsing stage.

The first parsing stage, which is accomplished by the RPL system function EXECUTE, classifies the RPL command. ALL RPL commands with the exception of the command (done) can be classified as shown in Figure V-3. The utility function POSIT scans the command line and returns the postion of the atom '=='. If '==' is not part of the command line POSIT returns 0. Using this information, combined with checking the length of the command line, syntax is verified and the parse is guided to either the function DEF_BINDINGS or DISPLAY for every command except the 'done' and 'file' commands. The 'file' and 'done' commands are directed to the FILE_WRITE and EXIT RPL systems function respectively.

The function DEF_BINDINGS, expecting one of the first three input forms shown in Figure V-3, completes the parse by checking the length of the command line. Knowing the length of the command, the name and expression can be

extracted using the CAR and CDR functions. Once the expression is isolated it can be evaluated by calling the function EV.

If the expression is evaluated successfully several events occur. First the CONS function binds the name to the evaluated expression, and this pair is consed onto the RPL environment, E. Second, the name is consed onto the command

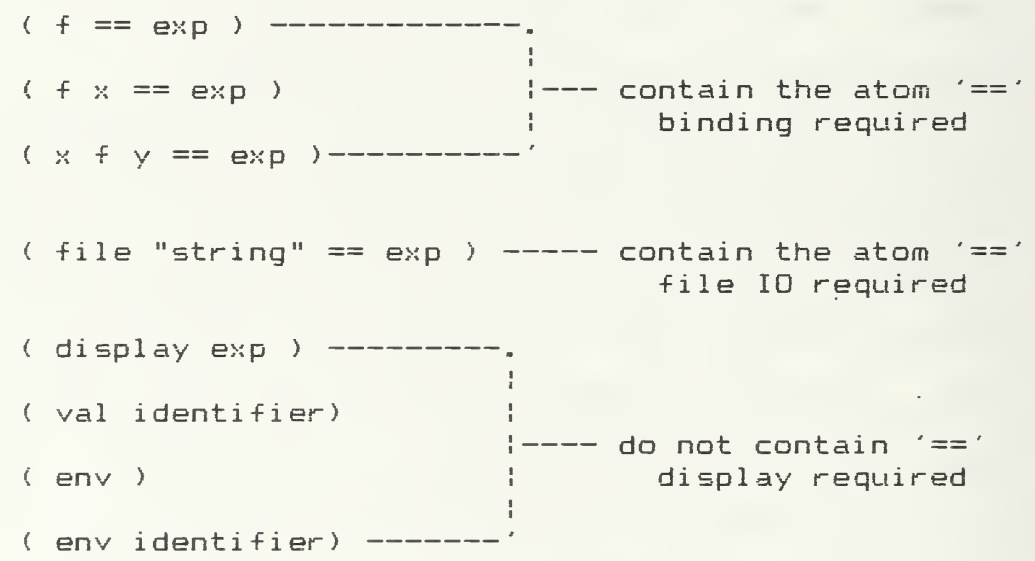


Figure V-3 -- Command line analysis

line and is added to the a-list called USERDEFS (giving the user the ability to save his commands to a file; see Chapter VI). Finally, if the binding is being made to a function defined using prefix syntax, the name of the

function is added to the global PREFIX_OPNAMES. If an error is detected while evaluating the expression the message 'BINDING CANNOT BE MADE' is given.

The DISPLAY function looks at the CAR of the command line to continue the parse and determine what must be displayed. If the 'display' command is used with an identifier, the name is looked up in USERDEFS and the command that generated the binding of that name is displayed. Otherwise the expression is evaluated and the result is shown. The debugging commands illustrated in the last three forms in Figure V-3 are also handled by the DISPLAY function (see section H on I/O).

D. EVALUATING RPL EXPRESSIONS

The heart of the RPL language is the expression. The expression is the vehicle that allows programmer's creative ability to be transmitted through RPL into something meaningful to LISP. The process of evaluating these expressions is centered around the RPL system function EV. This function, along with several auxiliary functions, parse and evaluate the expression recursively. The basic mechanism implemented by RPL used the design illustrated by MacLennan [Ref. 5: chap. 11] and Winston [Ref. 7: chap. 23], where a LISP interpreter was written in LISP, as a model.

There are two main differences between the design of the text book model and the RPL system. Every operator

implemented in the model design was in prefix notation. RPL must handle both infix and prefix operators and be able to recognize infix operators used with prefix syntax. The RPL system treats any infix operator as syntactic sugar for a prefix operator, which is made explicit in the use of the (op f) syntax. In this respect, the RPL system design is much more complex than its model. Adding to RPL's complexity was code necessary to provide a robust interpreter that would survive common programming errors. The error detection/recovery mechanism is discussed separately in section G.

The remaining section will explain the design of the EV function and its auxiliary functions that together provide the mechanism to evaluate the RPL expression.

1. The EV Function

EV is a function which was named after the LISP function EVAL, since functionally EVAL and EV are identical. Every expression in a LISP program is sent to EVAL. Every expression in an RPL session is sent to EV. EV, then, is the single most called function in the system. It takes two parameters, a RPL expression and a pointer to the environment of evaluation, which is the global environment when called originally. Using indirect recursion, EV and its supporting functions provides an effective mechanism which is central to the power available in RPL.

The case analysis shown in Figure V-4 provides the framework for the design of EV. The RPL expression is represented in LISP as either an atomic entity or as a list. Both these cases can each be further subdivided into three possibilities. Using the LISP conditional, COND, the logic suggested in Figure V-4 can be encapsulated into one efficient statement. COND is efficient since it stops evaluation at the first true statement. By carefully ordering the possibilities shown in Figure V-4 the number of unsuccessful checks can minimized. The order shown in

TYPE EXPRESSION	EXAMPLE	EV ACTION
LISP atomic		
numeric	5	return 5
string	"hours"	return "hours"
literal	avalue	call LOOKUP
LISP list		
special syntax	(if p -> f ; g)	call EV_SPECIAL_CASES
length 2	(not p)	call PREFIXOP
length 3	(x + y)	call INFIXOP
list w/bar	(f (, bar) g)	call EV_SPECIAL_CASES

Figure V-4 -- Case Analysis for EV

Figure V-4 is considered the most efficient since for every call to EV with a prefix or infix expression, which is a list, will require 2 or 3 calls to EV to evaluate the operator and the operands. In most cases these will all be atoms. Having the atomic values checked first, since they will be the operand to EV 2 or 3 times more often than the lists, takes advantage of LISP's implementation of the COND function.

A numeric or string value sent to EV is immediately returned since these values are the same in RPL as they are in LISP. The Literal atom, which is used to represent any of the RPL primitive data types, when sent to EV, must be found in the environment so that the value to which it is bound can be returned. This value is obtained by calling the RPL system function LOOKUP with the variable name and the environment pointer (see Figure V-4). If the variable is not found, NIL is returned from LOOKUP, which will trigger an error in EV.

When the expression sent to EV is a list it may have special syntax that requires special handling. Most cases are identified by a distinguishing tag in the grammar: 'op', 'lsec', 'rsec', etc. These tags are listed in the global variable SPECIAL_CASES. If the CAR of the expression is found in the list of SPECIAL_CASES the expression is sent to EV_SPECIAL_CASES for evaluation. Otherwise, the length of the expression becomes the key to its disposition. This is possible due to the modifications that were made to 'lispify' the grammar (see Appendices A and B). Prefix expressions are of length 2 from the production

expression -> (application primary),

while infix expressions are of length 3 from the production

expression -> (expression infix expression).

With this information EV can call either PREFIXOP or INFIXOP

to finish the parsing and continue the evaluation process on the expression.

There is one exception to the method just outlined. Before calling INFIXOP one final check must be made for special syntax to detect the use of the 'bar' with an infix operator. This syntax is used to combine functions. The following expression

```
(f (+ bar) g)
```

is a function represented by

```
(closure x ( (f x) + (g x) ) Ep).
```

This closure is created in EV_SPECIAL_CASES.

The following subsections will illustrate how RPL internally translates an infix to a prefix expression, in order to maintain a single internal application function and provide a high degree of user flexibility. The four step mechanism to perform functional application will also be discussed. The process includes:

(1) the evaluation of the actual parameters

(2) binding the formal parameters to the actuals to form the local environment

(3) the addition of the local environment to the function's environment of definition creating the evaluation environment

(4) the evaluation of the body of the function in its evaluation environment.

This application process is the key to the power of RPL.

2. Evaluate Operands - PREFIXOP and INFIXOP

These two functions provide the next level of parsing required to determine the semantics of the expression. Both functions are called from EV with a RPL expression and an environment pointer. The operator and its operands are extracted and calls to EV are made to ensure operands are defined and the operator is defined as a function. Completing these checks, the first step in the application process is accomplished. Note that no validation of operand compatibility with the operator is done at this time. If no errors have been encountered, the process continues. This is where INFIXOP and PREFIXOP differ slightly.

Since the expression in PREFIXOP has the syntax needed by the RPAPPLY function, where the application process continues, no further processing is required in PREFIXOP. However, since RPAPPLY must handle both prefix and infix expressions, before calling RPAPPLY INFIXOP must convert its operands into a two element RPL list. Therefore, if L and R are the evaluated arguments of the expression originally sent to INFIXOP, the parameter sent to RPAPPLY will be the equivalent to the RPL list (list L R). This would have the follow internal representation:

(Erel (1 L) (2 R)).

In summary both PREFIXOP and INFIXOP can be considered preprocessors for RPAPPLY. In addition, by

evaluating the operands, they perform the first step of the functional application process by evaluating the operands.

3. Binding Formals and Evaluation - RPAPPLY

RPAPPLY has one primary task, to complete the functional application process. To do this it first must determine whether the function being applied has been implemented in LISP directly as part of the RPL kernel. The kernel functions are readily distinguished from user or built-in extensional functions by the length of the list containing the function's definition. For example the function '+', which is implemented in LISP directly, is bound to

(closure +)

in the environment. The function DELTA is extensionally defined and bound to

(closure x (list x x) EP).

RPAPPLY passes all built-in functions that have been coded as part of the kernel (length 2 closures) to BIF_APPLY (restrictive relative closure). For user and extensionally defined functions RPAPPLY completes the functional application process recursively through EV.

The arguments to RPAPPLY are the products of either PREFIXOP or INFIXOP. The function and the actual parameters have both been evaluated. To complete the application process the function's formal parameters, body and

environment pointer are extracted from its definition (closure). Formals are bound to the actuals by using the CONS function to create the local environment. The number of formal parameters must match the number of actuals. If no error is detected, the local environment is consed onto the environment of definition creating the evaluation environment. With this new environment the function body, which is a RPL expression, can be evaluated. This requires another call on EV. Thus recursion is used indirectly to make a very powerful evaluation mechanism.

The following example demonstrates the way RPAPPLY completes the functional application process. Suppose RPAPPLY is called with the following arguments:

F <= (closure x (x + 1) Ep-f)

A <= 8

Since F is of length 4, RPAPPLY knows this is not a LISP coded function. The local environment, LE, is constructed,

LE <= (CONS X 8) = (x . 8).

The evaluation environment, EE, is constructed,

EE <= (CONS LE EP-f) = ((X . 8) EP_f)).

Now EV is called to evaluate the body of the function,

(EV '(x + 1) EE).

4. Built-in Functions Are Handled By BIF-APPLY

Of the 112 RPL operators, 68 are coded directly in LISP. These 68 functions form the kernel of RPL and 54 are handled in BIF_APPLY. The other 14 operators have unique

syntax and are handled in EV_SPECIAL_CASES. The parameters to BIF_APPLY are the same as to RPAPPLY: taking both a function and its argument in evaluated form. In the case of infix operators, operands have to be extracted from the argument list.

As discussed in the previous section the functions which are coded directly in LISP are bound to a definition represented by a list of length two. The second element of this list is used as the key to a very large LISP conditional. To find this key the conditional is divided into two logical parts, the built-in infix operators followed by the prefix built-in operators. Since all the built-in infix operator names are listed globally in the list BIFTAG_INFIX, checking for membership in this list directs the function to the appropriate section of the conditional.

Once the key has satisfied one of arms of the conditional, operand compatibility is verified. If no errors are detected the code which implements that operator is executed. Otherwise, an error handling mechanism is triggered which will provide both meaningful diagnostics and a graceful way of unwrapping the process back to the RPL command mode. RPL error handling is discussed in detail in a later subsection.

This huge nested LISP conditional can be considered the end of the line for any recursion that might have been

necessary through the application process. The result of this function will find its way back to EV through RPAPPLY and either PREFIXOP or INFIXOP.

5. Special Syntax - EV_SPECIAL_CASES

From an RPL programmer's perspective RPL is a language with an enormous flexibility. Much of the programming power in RPL is achieved through the use of special syntax to create programs mathematically. Although RPL has 70 operators implemented in the kernel and 45 extensionally defined, the language technically has an infinite number of operators available to the programmer. This power and flexibility is achieved through special RPL syntax. EV_SPECIAL_CASES is called from EV to evaluate expressions that have the atom 'func', 'op', 'lsec', 'rsec', 'if', 'bar' and 'iter' in them. In addition, EV_SPECIAL_CASES provides a mechanism to distinguish between the input and internal forms of sets, relations and RPL lists and always returns the internal evaluated form.

From an implementation perspective, EV_SPECIAL_CASES became a trap for cases that did not really fit anywhere else syntactically. This was particularly useful in the implementation of the 'if' and 'iter' operators. These both return a closure.

The implementation strategy for all special cases whose outcome was a function was the same. Each closure is created by parsing the expression to capture the semantics

of the expression within the new body of the new function. The body is another RPL expression. For example, the expression

(1sec + 1)

would be translated into a closure of the form

(closure ?x (?x + 1) Ep).

This methodology was adopted to implement 'if' and 'iter'. To preserve the semantics of the original expression, special syntax was introduced for the body of the closure, which would be special cases not available to the user. Adding these expressions to the lists handled by EV_SPECIAL_CASES provided the facility to capture the semantics of these expressions. The following example will illustrate the translation that occurs whenever 'if' and 'iter' are used:

(if p -> f ; g)

becomes

(closure ?x (when (p ?x) do (f ?x) elsedo (g ?x)) Ep),

and

(iter p -> f)

becomes

(closure ?x (repeat f untilnot p)) Ep).

By adding 'repeat' and 'when' to the list of special case tags these new syntax forms can also be evaluated by EV_SPECIAL_CASES, where they are parsed and evaluated directly in LISP. Note that the 'repeat' syntax above shows

the initial condition for the iteration. The result of evaluating (f ?x) becomes the agrument to 'p'. If the predicate is true, 'f' is evaluated with the result of the 1st iteration as an argument. This process continues until the predicate fails. The result of the iteration is the last value of (f ?x). This is all done in the REPEAT RPL function.

The rationale to create new RPL expressions for system use only was so successful, it became apparent that implementation of other operators like 'red', an array reduction operator, could use the same convention. Since 'red' is an infix operator and has no special syntax, a slight conceptual problem of where to create the closure emerged. All closures that were formulated thus far were done in EV_SPECIAL_CASES, but these came from cases having special syntax. Since the 'red' operator had no special syntax, it was inappropriate to create the closure in EV_SPECIAL_CASES. To be consistent with the design, the closure was created in BIF_APPLY. However, in formulating the body of the closure a special syntax is used which can be identified and evaluated readily by EV_SPECIAL_CASES. The closure created for this operator is illustrated by the following example: the expression

(f red i)

becomes

(closure ?A (reduce ?A by f from i) Ep).

Another choice of implementation would still result in the final evaluation in EV_SPECIAL_CASES but would eliminate a call to BIF_APPLY. Since what is actually hard coded in LISP is the function ARRAY_REDUCTION, the 'red' operator could be defined extensionally in terms of the special syntax and take advantage of the bindings created by RPAPPLY. For example, if the 'red' operator were defined

red == (func (f i) (func ?A (reduce ?A by f from i)))

or

f red i == (func ?A (reduce ?A by f from i))

the call to EV from RPAPPLY with 'red' and the environment Ep would produce the same results as what was accomplished by BIF_APPLY. However, this implementation would require the error checking now done in BIF_APPLY to be shifted into the function ARRAY_REDUCTION.

Tracing the evaluation of the extensionally defined 'red' shows the subtle differences between implementations. Given the expression

(f' red i'),

EV recognizes the infix expression and calls INFIXOP, where the 'red' is evaluated. The current implementation calls BIFAPPLY since the evaluated form of 'red' is

(closure reduction).

However, in the extensionally defined implementation 'red' is bound to

(closure (f i) (func ?A (reduce ?A by f from i)) Ep).

In the current implementation BIF_APPLY is called to finish the application process directly in LISP, whereas the alternate implementation uses the mechanism provided in RPAPPLY. The formals, f and i, are bound to the actuals f' and i'. The evaluation environment is created and EV is called to complete the process with the expression

(func ?A (reduce ?A by f from i) EE).

The 'func' tag directs the expression to EV_SPECIAL_CASES where the closure is created.

The difference in implementation efficiency can be studied by using the LISP function BREAKDOWN. Currently the composition and paralleling operator are defined using the 'func' construct as extensionals.

F. EXTENSIONAL MECHANISM

Almost half the operators in RPL have been implemented extensionally. The operators directly coded in LISP either were listed as primitive operations by MacLennan [Ref. 2] or had a function readily available in LISP which would hopefully provide a more efficient implementation than the extentional definition. The purpose of this section is to discuss the mechanism which the system uses to implement an extensional operator.

The extentionally defined operator is executed by the RPL system taking advantage of the same mechanism that is in place to bind user defined functions. When RPL is called,

after all the globals have been initialized, (see Figure V-5), the system is ready to define the extensional operators.

During initialization the environment contains all the built-in operators which are coded in LISP. These are represented by a length 2 closure as discussed earlier in this chapter. At this time commands can now be accepted by the system. All the extensionally defined operators are contained in a list as RPL commands. This list is called INTOPS. Using a the LISP function MAPCAR, all extensional commands are sent to EXECUTE and ultimately bound to the environment. After the last extensional operator has been defined the system ready for the user.

Implementing this mechanism was straight forward but there were some varibles that had to reset before the user was given control of the system. These are shown in Figure V-5.

The interesting part of this implementation was the ability to try each of these operators during RPL sessions prior to committing them into the list of extensionals. Since some extensionals were built on others, the order that these were actually defined was significant. This was due to static scoping. Therefore, some care had to be used when adding new definitions to INTOPS. Future research may try coding some of the extensionals of this implementaion into LISP directly and do a performance analysis using BREAKDOWN. This will be discussed in more detail in Chapter VII.

NAME / INIT VALUE	CHANGED BY	PURPOSE
BIFTAG_INFIX / list of names	N/A	Control flow in BIF_APPLY
BUILT_IN_PREFIX_OPS / list of names	N/A	Resets PREFIX_OPNAMES
E / SYSOPS	DEF_BINDING EXIT	Global environment
EMSG / list of msgs	N/A	Table of error messages
ERRORCODE /. ERRORFREE	ERROR_HANDLER	Error recovery
FILTER_ON / NIL	FILTER	Switch off error msgs while filtering
INTOPS / list of commands	N/A	All extensional operator definitions
NUMOP / list of operators	N/A	Control flow in BIF_APPLY
OPNAMES / list of names	N/A	Check to avoid renaming built-in ops
PREFIX_OPNAMES / BUILT_IN_PREFIX_OPS	DEF_BINDING EXIT (reset)	Error checking for 'op', 'lsec', 'rsec'
SETS / list of input tags	N/A	Control flow in EV_SPECIAL_CASES
SETOPS / list of operators	N/A	Control flow in BIF_APPLY
SYSOPS / list built-in operators	N/A	Kernel
SYSTEM_ENV / E	N/A	To reset E when clearing environment
USERDEFS / NIL	DEF_BINDING EXIT (reset) RPL (reset)	display WRITE_USER_DEFS

Figure V-5 -- Alphabetic Global Listing

G. ERROR DIAGNOSTICS AND RECOVERY

The primary consideration for performing error checking in the RPL interpreter was to ensure the system would survive common programming errors. If every minor miscue were to cause the RPL system to crash errors like undefined variables, improper arguments to built-in and user defined functions, syntax, spelling and typographical errors, each could cause a major catastrophy, costing many hours of work and added programmer frustration. Surely a system without safeguards to prevent self destruction would be impossible to work with, even in a prototype implementation. Therefore, one of the major design decisions in the development of the interpreter was to make the RPL system as robust as possible and provide meaningful diagnostics to the user.

1. Error Recovery

LISP's built-in functions are not unlike those found in any other language; improper operands are generally a disaster. A keen awareness of this problem had to be developed to ensure sufficient type checks were accomplished so that user inputs could not create an unrecoverable situation. Although Interlisp does provide a means of error recovery through its debugging facilities, this is only a benefit to the user who has had sufficient experience with the Interlisp break commands (see Teitelman [Ref. 6] for more details). Therefore, it was necessary to build into the RPL system a self-contained capability that could detect,

diagnose and resume operation, totally independent from the LISP error handling mechanism.

Once an error is detected, the RPL system calls its error handling function with two parameters. The first parameter is an error code, which is used as an index to a table of error messages. The second parameter is the cause of error. The error handler prints the appropriate error message and cause of error, and assigns to the global variable ERRORCODE the value of the first parameter. ERRORCODE is always initialized to ERRORFREE before a command is entered by the user. Finally, the value returned by the error handler is the LISP atom NIL.

Checking the value of ERRORCODE in strategic areas throughout the program prevents both redundant error messages and meaningless operations. For example, in the process of evaluating a prefix expression both the operator and the operand must be evaluated separately to ensure they are defined. If any errors are encountered in this process the remaining code in the prefix expression parse can be by-passed by checking the value of ERRORCODE before preceding.

The value of ERRORCODE is checked before any bindings are made to the RPL environment. If ERRORCODE is not ERRORFREE the message 'Binding cannot be made' is given.

The value of the functions that parse either prefix or infix expressions each return NIL if an error occurs. In the RPL DISPLAY function, if the LISP value NIL is returned

from evaluating an expression, the message 'Undefined' will be displayed.

Calls to the error handler and the inspection of the value of ERRORCODE is interwoven throughout the RPL system. This was impossible to avoid, if the RPL system was to have the degree of resiliency desired. To change the basis of the error handling mechanism used would certainly take a considerable amount of recoding. This should be unnecessary due to the excellent recoverability shown in the RPL system during testing.

2. RPL Diagnostics

RPL suffers from a problem prevalent among many extensible languages, its diagnostics are sometimes meaningless. This is because error checking is performed on the operands of the functions defined in the kernel of the language. The kernel is a set of functions from which additional features are implemented. The diagnostics related to calls on these functions, when used explicitly by the user, are helpful and descriptive. These same diagnostics, when given to a user who is invoking a function defined in terms of the kernel, may be of little or no value.

The diagnostics displayed when an error is discovered while performing a domain restriction illustrate a situation where the system can provides accurate but confusing diagnostics. The operator '->' is defined directly in terms of the RPL's 'filter' operator as follows:

```
p -> t == ((p o hd) filter t).
```

The composition operator 'o' is defined using the formal parameters f and g as follows:

```
o == (func (f g) (func x (f (g x)))).
```

A user who is unfamiliar with these dependencies would certainly find diagnostics in terms of p, t, f or g quite puzzling if he had never bound these names in his environment.

The more familiar that one becomes with the RPL system and the various extended functions, the more meaningful the diagnostics will become. When given diagnostics that appear totally unrelated to what was input as an RPL command, there is an excellent possibility that an extended operator is being used. Probing the environment with some of the features added to RPL as a troubleshooting aid (env, val and env f) will help put more meaning into error diagnostics, and enable the user to better understand the RPL language.

3. Errors Can Be Easily Built In

The incompatibility between functions and their arguments referred to thus far are a direct result of user errors. Guarding against this kind of circumstance was only part of the problem encountered to make RPL robust. Extreme care had to be taken not to build potential fatal errors into the interpreter. This became apparent as the system

crashed in areas originally thought to be sound, during some of the earlier RPL system tests.

As discussed in Chapter III, the functions CAR and CDR are used to access various elements of a list. Like any other function, these functions are designed for a specific type of operand. Calling either function with a non-list creates a fatal error. The system was vulnerable to this situation in the original coding. To prevent this type of error, each time the CAR/CDR functions appeared in the development of the interpreter, a list check and/or length check had to performed before proceeding. The code used to implement the type checking function, TYPE, for the RPL system indicates the caution needed when using these functions. This is also evidenced the use of compound statements in many LISP conditionals, where the AND statement first performed a list check and then a length check before using the CAR or CDR functions.

Achieving the goal of making RPL robust involved much more than an exercise in anticipating user errors. It also required a conscientious analysis of every aspect of the interpreter to determine what inputs or results could create disaster. Testing thus far has shown that this goal has been essentially achieved.

H. INPUT / OUTPUT

The input/output functions needed by the RPL system can be logically divided into two categories, console IO and file IO. Console IO functions provide a mechanism to input RPL commands, display the results of evaluating RPL commands, provide error messages and prompt the user for input. The file IO functions provides both a facility to execute the RPL 'file' operator and gives the user the ability to save and recall his RPL sessions.

1. Console Input/Output

The primary consideration for altering normal LISP IO originally was the aesthetic desire to eliminate parentheses not absolutely essential to parsing and command execution. This is achieved through masking some of the required input parentheses and filtering meaningless parentheses during output. This eliminated some of the awkward syntax that had been introduced in order to use LISP as an implementation language (see Why LISP Chapter III). As the interpreter developed, a far more important reason for filtering console output was realized.

The only relief from the LISP syntax during terminal input was achieved by the elimination of the outer set of parentheses from the RPL command line. This was accomplished through the use of the Interlisp READLINE function. This function inserts parentheses around a line of input which is terminated by a carriage return or the character ']'.

Additionally, the READLINE command provides a mechanism to enable the user to know when all open parentheses have been closed. This is illustrated in the following example.

If the user wants to type the command

```
(f x == (x + 1)),
```

the READLINE function would allow it to entered as follows:

```
f x == (x + 1).
```

When the user types the closing parenthesis after the '1', the the following would be displayed:

```
f x == (x + 1)
...
```

The ''...' indicates all parentheses have been closed. A carriage return at this point will enter the command for execution. Since every RPL expression must be enclosed in parentheses, this feature is particularly helpful to the programmer.

To 'delispify' RPL output, user prompts and error messages were printed by a function written to filter parentheses by printing lists one atom at a time, using the very fast and efficient LISP MAPCAR function. This methodology was originally used for all RPL output, but had to be restricted to prompts and messages. This restriction was necessary since the way lisp prints a list proved to unsuitable for printing the internal definition of a function. This problem was encountered printing output from the RPL 'display' command.

The method chosen to internally represent functions made displaying them on the screen impractical and in some instances impossible.

As discussed in Chapter V, each function that is either user defined or built-in as an extension of the RPL kernel, has associated with its name the keyword 'closure', its formal parameters, its body and its environment of definition. This environment, which is represented as a pointer to an a-list in LISP, includes all RPL built-in functions along with all names and functions defined by the user up to time the function was defined. Printing this environment had to avoided. This was accomplished by creating two integrated functions, PRINT_LIST and SHOW_ATOM, to screen all RPL output, trapping all functions so that the environment could be truncated for console output.

To maintain the user's ability to inspect the environment, some additional features had to be added to the RPL system. This resulted in a minor modification to the grammar and the addition of the function SHOW_ENV. For example, typing 'env' provides a list of all names with their respective internal definitions that are within the environment created by the user. Each function, of course, would be shown without its environment of definition. To display the environment of definition associated with a given function f, the command 'env f' is used.

Two additional features were also added that allow the user to view either the internal definition associated with a name or his original input form. This is discussed in more detail in Chapter VI.

2. File Input/Output

There are two sets of file IO functions used within the RPL system. The first set, consisting of the functions FILE_READ and FILE_WRITE, is used to implement the RPL 'file' operator. The second set, added as a user convenience to provide a mechanism to save and recall RPL sessions, is comprised of the functions SET_USER_ENV, READ_USER_DEFS, EXIT AND WRITE_USER_DEFS. Both sets of file IO functions utilize the Interlisp file package commands to access or initialize a file, perform desired IO and close the file.

RPL's 'file' operator is designed to read or write data in its evaluated form. This data is usually a set or table. This operator should never be used with functions, either directly or indirectly, embedded within a set. This would cause the function's entire environment of definition to be written to a file as a list, one atom at a time. Reading a function from a file that was written in evaluated form, not only may be impossible due to insufficient memory, but obviates the efficiency of the environment mechanism. RPL was designed to have only one a-list represent its environment. A function's environment of definition is just a pointer to a node within the RPL system environment.

In a typical RPL session a user may have a considerable amount of time invested constructing numerous functions and data definitions. As a command is entered that binds a name to the RPL environment, the command is saved in a separate list that can be written to a file. When read back into RPL, the system executes each command, thus recreating the previous session.

The user has the flexibility to modify or create files using any available editor. His only constraint is to ensure the string EOF appears as the last line of the file. The EOF string is automatically written to all sessions saved in RPL.

Interlisp operating on UNIX provides a means to save old versions of files as new files are created. The updated file will have its file name modified to indicate the next version number. Since UNIX only recognizes unnumbered names, each updated file created by Interlisp contains two directory entries, one numbered and one unnumbered. Interlisp provides the mechanism to manipulate older versions [Ref. 8: p. 11].

VI. USING THE RPL INTERPRETER

A. INTRODUCTION

The RPL language is different from any conventional language that currently exists. Because of its uniqueness, inherent power, and mathematical base, it can be difficult to use at first. But, as with any other language, it can be mastered through a study of the underlying concepts and hands on experience with the commands. This chapter will describe the basic knowledge required to use the prototype RPL interpreter developed in this research. It will only touch upon, through simple examples, the power of such a language. Only the dedicated efforts of an innovative user will test the system and discover the real potential of the relational programming concept.

B. GETTING STARTED

The RPL interpreter exists as a Unix file which consists of 77 LISP functions which implement the RPL grammar shown in Appendix B and the relational operators described in Appendix C. To invoke the RPL interpreter, a user must first have a basic knowledge of the Unix Operating System. He must at a minimum be able to log on with access to an account which contains the 'RPL-INT' file. For more information on the Unix Operating System, see reference 8.

When the Unix prompt (%) appears, the next step is to enter the Interlisp environment, which provides a shell for RPL. Since the interpreter is written in LISP, familiarity with its basic constructs is desirable, and a necessity if one is going to explore the LISP code for the interpreter itself. See references 5, 6 and 7 for more information about LISP and the Interlisp environment.

Loading the Interlisp environment is accompanied by a substantial delay, but when the environment is finally loaded, it gives the user a friendly greeting to let him know it is ready to accept commands. The only LISP command that must be used is 'LOAD' which loads a file(s) of LISP functions. Therefore, at the LISP prompt, '_', the user must type 'LOAD[RPL-INT]'. When the closing bracket is typed Interlisp will automatically execute the command. Interlisp searches the user's directory for this file and, when it is found, displays a message indicating the date the file was created. Once loaded, another Interlisp prompt will be displayed. Now all the functions necessary to execute RPL commands are part of the Interlisp environment, but of no use to the programmer until he invokes the RPL interpreter itself.

All commands in LISP are enclosed in parentheses or brackets. Just as the keyword 'ilisp' triggers the Unix system to load the Interlisp environment, the LISP function 'RPL' initializes and loads the RPL environment on top of

Interlisp. Thus to begin an RPL session the LISP command '[RPL]' is typed at the LISP prompt. Once this command is executed, the user will enter and remain in the RPL environment until the RPL command 'done' is executed. (See section I for exceptions).

When the initialization required by the RPL interpreter is completed the user is asked if he wishes to resume a previous RPL session. This gives the programmer the option of having a file of RPL definitions executed that was created either from within RPL or by an external text editor. Caution is advised if the file was created by an external editor since no error checking will be done until loading such a session for the first time. If there is a parenthesis out of place or missing, it could throw the user out of RPL and into the LISP error handler. Some other dangers are discussed in section I of this chapter.

If the user answers 'yes' he will be prompted for a filename. It is appropriate to mention at this point, that an inconvenience exists due to the limited control over input/output by the interpreter. When a response is required, or a command is entered, the first character typed is fixed, i.e., it cannot be removed from the input buffer. All characters after the first one can be altered as required until a input termination signal is sent. In the RPL environment, hence the Interlisp environment, this signal is a carriage return or a final closing parenthesis

or bracket. Thus if the user makes an error, for whatever reason, and the filename is not in his directory, RPL will inform the user that the file was not found and continue on. The only avenue open to the user if this happens, is to terminate the session and begin again. This is not as bad as it may sound, as the next section will point out.

If the file does exist, RPL will load and execute all commands in the file, and prompt him for his first RPL command. Figure VI-1 and Figure VI-2 illustrate the command sequence which would load RPL with and without a previous session, respectively.

C. SESSION TERMINATION

When the user is finished with a session he types the command 'done'. This command triggers a series of options available to the user. First, he will be asked if he wishes to save the session just completed. If the answer is yes he will be prompted for a filename. RPL will write all commands executed in the session, in their original input form, to that file. 'Display' commands are not included.

Regardless of his answer to the first question, the user is then given three options: exit to the Interlisp environment, exit to the Unix Operating System, or begin another RPL session. If he chooses to begin another session he will be asked if he wants the current environment from the session he is leaving to be cleared. After completing

```
% ilisp
ISI-INTERLISP 15-MAY-84 ...

Hi.
_LOAD[RPL-INT]
File Created:12-MAY-85 10:03:30
RPL-INTCOMS

expanding LISTP, 65520 used, 2424832 before GC
/work/brown/RPL-INT
_[RPL]
Loading RPL--- DO YOU WANT TO RESUME A PREVIOUS RPL SESSION? <y/n> y
INPUT FILENAME
sess512
Loading--- Session loaded


RPL INTERPRETER ON LINE!!

?> d (2 + 3)

 5
```

Figure VI-1 -- Loading the RPL Interpreter, With Previous Session

the action required by the user's response, the RPL interpreter begins the same cycle as if the user was beginning a new session. This cycle continues until the

```
% ilisp
ISI-INTERLISP 15-MAY-84 ...

Hi.
_LOAD[RPL-INT]
File Created:12-MAY-85 10:03:30
RPL-INTCOMS

expanding LISTP, 65520 used, 2424832 before GC
/work/brown/RPL-INT
_[RPL]
Loading RPL--- DO YOU WANT TO RESUME A PREVIOUS RPL SESSION? <y/n> n


RPL INTERPRETER ON LINE!!

?> sqr x == (x times x)

?>
```

Figure VI-2 -- Loading the RPL Interpreter, Without Previous Session

user decides to completely exit the RPL environment. Figure VI-3 illustrates a session termination sequence where the user wishes to remain in the RPL environment. Figure VI-4 shows a user termination with exit to the Unix Operating System.

D. EXECUTING COMMANDS

RPL commands are derived from the grammar in Appendix B. It allows for three basic types of commands: data

```
?> done

DO YOU WANT TO SAVE ENVIRONMENT FOR FUTURE USE? <y/n> y
INPUT FILENAME
sess525

EXIT TO LISP - PRESS ^D
EXIT TO UNIX - PRESS ^C
CONTINUE RPL - PRESS <RETURN>

DO YOU WANT TO CLEAR CURRENT ENVIRONMENT? <y/n> y

DO YOU WANT TO RESUME A PREVIOUS RPL SESSION? <y/n> n
?>
```

Figure VI-3 -- Session Termination - Remain in RPL

```
?> done

DO YOU WANT TO SAVE ENVIRONMENT FOR FUTURE USE? <y/n> y
INPUT FILENAME
sess525-1

EXIT TO LISP - PRESS ^D
EXIT TO UNIX - PRESS ^C
CONTINUE RPL - PRESS <RETURN>

% logoff
  Signing off...
```

Figure VI-4 -- Session Termination - Exit to Unix

definitions, function definitions, and input/output. The sections following this one will describe how to enter the commands of each type and provide a brief discussion of the built-in relational operators. This section will provide some general information and guidance.

RPL operators and commands are case sensitive. Since most operators and all commands are in lowercase, it is recommended, though not required, to use lowercase letters throughout an RPL session. Lowercase was used to help distinguish the operators and commands from LISP function names, which are capitalized. Any variation at the keyboard will cause RPL to return an error.

E. DATA DEFINITIONS

1. Introduction

There are several data types available to RPL. In addition to the normal scalar types, integers, reals, booleans and strings, there are sets and relations. Sets and relations can be used to represent any conventional data structures such as arrays and records. They can also easily represent more complex structures such as matrices, databases, trees and graphs. A relation is actually a special form of set where each element must be a pair of RPL data types. The tremendous flexibility of the relation results because this pair can be any combination of RPL data types.

RPL syntax allows the binding of a name to any scalar type directly. For example:

```
x == 3
error == true
name == "John"
```

The '==' symbol in RPL means 'defined as'.

2. Sets

A set is defined simply by placing the keyword 'set' as the first element of the set. For example:

```
aset == (set 1 2 "dog" colors]
```

The ']' symbol, used to close the defintion, keys the interpreter to execute the command. This aspect of the command line will be discussed further in section 6. Note the name colors must have been previously defined or an error will result. In this case, colors may have been defined as:

```
colors == (set "red" "white" "blue"]
```

This illustrates that each element of a set can be anything, even another set.

3. Relations

Any relation can be defined in RPL using the following syntax:

```
r == (rel (X1 : Y1) (X2 : Y2) ... (Xn : Yn)]
```

The X's and Y's can be any RPL data type. The ':' symbol is the pair-making operation. It binds any particular X and Y

together into a pair, distinguishing it as an element of the relation. Note that there must be a space on either side of the operator. This is required because this structure is treated internally as a LISP list. If a space is left out, an RPL error will occur.

To demonstrate the utility of this structure, a sequence, an array and a record will be defined below:

```
sequence == (rel (1 : 2) (2 : 3) (3 : 4)]
array == (rel (1 : "a") (2 : "b") (3 : "c")]
record == (rel ("#" : 101) ("name" : "John") ("age" : 32)]
```

Even more complex data structures can be formed easily by combining these and other primitive relational structures. For example, a database is just a set of records. Since there are so many different forms of a relation, RPL has included syntax to simplify the definition of two of the more common ones, the sequence and list.

4. Sequences

The relation 'sequence' shown in section 3 can be entered as:

```
sequence == (seq 1 2 3]
```

It must be pointed out that this is a pure sequence, i.e., a relation which has one initial element, one terminal element, and is fully connected. Formally, it is an irreflexive connected bijection. Graphically, this sequence can be represented as shown in Figure VI-5.

The label that is put on a node is not important, so the sequence, (seq 5 2 10 9), is equally as valid as the one

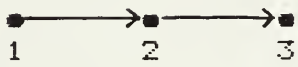
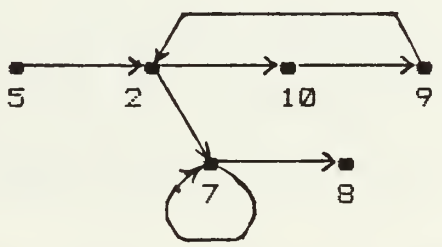


Figure VI-5 - Graphic representation of a sequence

shown in Figure VI-5. However, RPL does not prevent the user from entering:

sequence == (seq 5 2 10 9 2 7 7 8)

This is an invalid sequence and is represented graphically:



Therefore, it is up to the programmer to insure that he is defining a proper sequence. The sequence operators do not verify that the structure passed to them is a valid sequence. When this occurs, an error can result, the results can be meaningless, or at worse the computation may not halt - forcing the user to abort the session and lose everything. For this reason, caution is advised. On the other hand, the lack of rigidity in sequence definition permits the easy representation of certain types of directed graphs, as the example above points out.

5. Lists

The list is just a restricted form of an array which has a starting index of 1. An array, on the other hand, can have any integer as a starting index. The relation 'array' shown in section 3 is also a list and can therefore be written as: array == (list "a" "b" "c"].

The most common use for the list, and the reason it is included as a separate entity in RPL, is to represent argument lists. All multi-parameter functions in RPL are represented internally in prefix format and use the list as their argument.

6. Ranges

To simplify the data definition further when dealing with large numeric structures, the setrange, seqrange and listrange syntax is provided. For example, it is possible to define:

```
s == (setrange 1 to 50]
s' == (seqrange 1 to 50]
lst == (listrange 10 to 60]
```

These definitions evaluate to the appropriate internal forms: s would be a set of the integers from 1 to 50, s' would be a relation which relates each number with its successor, up to 50, and lst would be a relation which relates an index, starting from 1, to each value from 10 to 60. The utility of this syntax becomes apparent when one

thinks about what is involved if these structures had to be entered using the general relation syntax.

The input forms discussed in this section can be used effectively within the RPL interpreter to create any form of data required. Sometimes it may be more convenient to use the simpler sequence and list syntax than the more general relation syntax to define a desired data structure. For example, suppose the user wanted a five element array which contained even numbers starting with 2, and which was indexed starting with 10. Internally, the desired structure would look like:

```
(rel (10 2) (11 4) (12 6) (13 8) (14 10))
```

With the relation syntax the user would have to write:

```
a == (rel (10 : 2) (11 : 4) (12 : 6) (13 : 8) (14 : 10)]
```

He could achieve the same result by using the sequence to array operator, sa, which takes a sequence, and a starting index as arguments, and returns the appropriate array. Thus, he could have typed:

```
a == ((seq 2 4 6 8 10) sa 10]
```

Which method is easier must be decided by the user and depends upon his degree of familiarity with RPL. Note, however, that the second format has less parentheses and spaces to contend with!

F. FUNCTION DEFINITIONS

Although RPL contains a rich set of built-in operators, it could never include everything, nor should it, that a user could want. RPL is extensible and thus includes a mechanism for defining user functions. As illustrated in earlier chapters, there are three definition options: direct, prefix and infix. Most user functions can be defined using the simple prefix and infix syntax. For example, if the user had a need for a function which would add 2 to its input and square the result, he could write:

```
add2sqr x == ((x + 2) times (x + 2)]
```

For a similar, but more general function, which takes two arguments he can write:

```
x addsqr y == ((x + y) times (x + y)]
```

An alternate definition for addsqr could be written using the DELTA operator, which duplicates an argument, and the composition operator:

```
x addsqr y == ((times o DELTA) (x + y)]
```

A third, and even more formidable looking definition is given by:

```
addsqr == ((times o DELTA) o (op +))
```

The last two definitions introduce the flexibility of RPL by showing how complex functionals can be easily defined in terms of built-in and/or user defined operators.

There are some cases, however, where the prefix and infix definitional syntax will not meet the user's needs,

and therefore the direct method for defining functions is included. One utility of this syntax is its ability to define functions with any number of parameters. For example, say a function called addsub is desired. This function adds its first two arguments and subtracts the third. It can be defined via the direct method as:

```
addsub == (func (x y z) ((x + y) - z)]
```

This is just another way to write:

```
addsub (x y z) == ((x + y) - z)
```

Notice that the argument to these functions must be a RPL list with three elements. The advantage of the direct syntax over the prefix-type syntax is the ability of the 'func' definitional structure to be imbedded within another function. This gives RPL the same flexibility as LISP with its 'LAMBDA' expression.

This same function could be defined using the prefix syntax, but the user must be aware of how RPL extracts the actual values from the argument list in order to bind its formal arguments to the actuals. This extraction is done by use of the RPL 'sel' operator. Thus when given a table and a member of its domain, this operator will return the first member in the range related to it. Equipped with this knowledge and familiarity with the list structure, the user can also define addsub in prefix form as:

```
addsub x == (((x sel 1) + (x sel 2)) - (x sel 3)]
```

This form and the direct definition are equivalent and will

work equally as well, but it is obvious in this case that the direct method is much simpler and more understandable.

G. INPUT/OUTPUT

1. Screen Input/Output

All syntax presented thus far is for commands that will be typed at the terminal in an interactive session as input. Output at the screen is generated using the 'display' commands. To recall to the screen any definition, the user can type the word 'display' followed by the name of the entity he wishes to see, e.g.,

```
display array <CR>
```

Notice that this is the first time that the requirement for a carriage return, <CR>, has been indicated. This is because the definitional forms discussed earlier ended with a ']' which automatically triggers execution. For commands such as display, and those that are ended with a ')', a <CR> is required. Execution of the command above will display the definition bound to the name 'array' in the environment. For example, it might be:

```
array == (list "a" "b" "c")
```

The display command can also be used to see the result of a computation immediately, but once displayed, the result is lost because it will not be bound to a name. For example if the user types 'display (3 + 5)' , '8' will be shown. Thus 'display' can have any expression as an argument. To

simplify output to the screen, the word 'display', and two shorter versions, 'dis' and 'd', are optional. Thus, only the expression itself needs to be typed to display a result.

2. File Input/Output

Any data definition can be saved to a file for future use simply by typing:

file "table1" == t <CR>

This command assumes that t has been previously defined, e.g., as a table of squares for a finite range. To later read that table into another RPL session, the user can type:

tbl == file "table1" <CR>

Since file input/output is implemented as a special command, it can also be used directly in an expression. For example, the command '((file "table1") sel 2)' would return '4' for the table of squares mentioned earlier.

3. Debugging

The final form of output to the screen in RPL was implemented to assist debugging. Since a function definition can involve the composition of many operators, both built-in and user defined, cause-of-error messages might give a strange response. This happens because the cause of the error may be rooted in the execution of one of the internal component functions within the definition. Likewise, there will be times when the user passes an argument to a function, but it is rejected as the wrong type. On these occasions, it is nice to be able to probe

deeper into RPL. The 'val' and 'env' commands provide this mechanism.

The 'val' operator applied to any name will return the evaluated form of the definition bound to that name. Thus, if s is bound to the sequence (seq 1 2 3), typing 'val s', will return '(rel (1 2) (2 3))'. Similarly, for the function sum, defined as (x + y), typing 'val sum' would return '(closure x ((x sel 1) + (x sel 2)))'. Notice the environment of definition is missing. As discussed in earlier chapters, the environment is omitted due to its excessive length.

The 'env' command provides the mechanism to view the environments that are omitted from the display of functions in evaluated form. The environment is shown in definitional form. Thus, 'env' alone will produce all definitions created during the current session. Applying 'env' to a function name will produce all definitions visible within its scope. For example, the result of typing 'env' for a short RPL session might be:

```
f == (lsec (times o DELTA) img)
s == (set 5 6 7 8)
x sum y == (x + y)
arg == (list 2 4)
System Defined Functions
```

The last definition put into the environment is shown first. 'System Defined Functions' constitute all of the built-in

function definitions within RPL. Finally, using the same environment, typing 'env sum' would return:

```
x sum y == (x + y)
arg == (list 2 4)
System Defined Functions
```

H. RELATIONAL OPERATORS

In the RPL interpreter there are 112 built-in relational operators based upon the operations described by MacLennan in reference 2. All the operators implemented within the RPL system are discussed in detail in Appendix C and are broken down into classes based on both the number and type of arguments, and what they return.

The operators are a mix of first and higher order functions.. A first order function is one that has data for inputs and outputs. A higher order function is one that has a first or higher order function as either input or output. Since RPL has several higher order functions they are further separated into two classes: those which return a function, and those which have a function as an input, but return data.

Finally, there is a group of operators which are unique because of their special syntactic requirements or their special handling required in implementation. They are consolidated under the title of 'Special Operators'. They include the data definition operators, a conditional functional, an iteration functional, a function to compute

closures, the empty operator, and the 'bar' functional which gives any infix operator a special meaning.

Based upon the preceeding discussion, the operators are broken into 11 logical classes as shown in Figure VI-6. The Global class of operators include those which take anything as an argument(s), or in the case of 'hd' and 'tl', return anything. The Arithmetic and Logical operators parallel their conventional counterparts. The next five classes are derived from the type (form) of the relation involved. Finally, there are the two classes of higher order operators, and the special operators.

1. Global
2. Arithmetic
3. Logical
4. Set
5. Relation
6. Sequence
7. Array
8. Database
9. Higher Order - Return Function
10. Higher Order - Return Data
11. Special

Figure VI-6 -- RPL Operator Classes

I. BEWARE THE KEYSTROKE

1. Introduction

Unfortunately, because the RPL Interpreter is running within the Interlisp environment and the Unix Operating System, there are a few keystrokes which may cause unexpected results. Some keystrokes should be avoided, some

should be used with caution, and some can be used to the user's advantage.

2. The Control-D (^D) and Control-C (^C)

Pressing a ^D should be avoided. It will abort whatever LISP function is being executed, return the LISP prompt and wait for the next command. Since the RPL interpreter is invoked as a LISP command, a ^D will immediately abort the user's RPL session, discarding all work done to this point. Likewise, only more severe, pressing a ^C will abort both RPL and Interlisp and return the user to the Unix Operating System.

The ^D and ^C are used, however, as part of the RPL system to exit the RPL environment. They are options within the RPL 'done' command and should be used only in this context. In general the Control key should be left alone since there is no meaning associated with control characters in RPL, and they may cause Interlisp or Unix to do unexpected and probably unwanted things.

3. The Backspace Key

A second key to be avoided is the backspace key. For reasons not totally understood to date, pushing the backspace key causes Interlisp to invoke the LISP error handling package. A strange message appears on the screen, which looks something like 'broken below ØGETTY' and a ':' prompt will appear. Fortunately, this is not the kiss of death as was the ^D. Typing 'RETURN NIL' (in capitals) will

return the user back to where he was in RPL before hitting the backspace key. If another strange message appears followed by another ':' then the user probably hit the backspace key more than once. A 'RETURN NIL' must be typed for each time the backspace key was hit, and only then will Interlisp return the user to RPL in the place it left off.

There is one instance in which this keystroke becomes an advantage. It can used to temporarily leave the RPL environment to invoke any Interlisp feature. Of particular interest is the 'BREAKDOWN' package. This package allows the user to do performance analysis of the LISP functions used within the RPL interpreter. A more detailed discussion of the benefits of this package will be presented in the final chapter. This feature of RPL is of real interest to those individuals who are interested in further research with relational programming and the improvement of the RPL interpreter.

4. The Control-Z (^Z)

The final keystroke to be discussed is the least dangerous, and in fact has a positive utility. Hitting a Control-Z (^Z) will temporarily suspend whatever the user is doing and put him back at the Unix logon level. The user can then execute any Unix command desired, e.g., he could look at his directory to verify the filename of a session he wished to load. When he is finished at this level, he types 'fg' (lowercase letters only) and returns to the exact place

he had left off when he pressed the ^Z. Thus the programmer can take advantage of the facilities, flexibility and power of the Unix Operating System concurrently while executing an RPL session.

VII. CONCLUSIONS AND RECOMMENDATIONS

The primary goal of this prototype RPL implementation was to provide a mechanism for future research. Prototypes generally have a definite starting point, which is the theoretical work of its creator, the language developer. What marks the completion of the prototype is a design decision that must be made. Along these lines, one of the most difficult dilemmas facing this implementation was handling implementation improvements that became obvious as the development progressed. Without exercising restraint, implementation improvements can become an obstacle to timely completion. Unless specific performance criteria have been set as a system design requirement, and it can be determined that a particular mechanism of the system must be changed to meet this objective, improvements that become obvious to the prototype developer should be documented for follow on research. Focus on design issues can easily become blurred and transition between prototype and future research obscured as improvements that become apparent to the developer divert efforts from the original goal. Let the completion of the prototype be the springboard to enhancements and efficiency issues.

Future research on RPL was one of the primary considerations in this prototype, which, as discussed in

Chapter III, prompted its implementation in LISP using the Interlisp environment. Tools available in Interlisp were a powerful incentive that influenced the choice decision of the implementation language used for RPL. The cost of this decision, however, was more than anticipated.

Using the Interlisp programming environment can be a very frustrating experience to a programmer. Documentation available ([Ref. 6] and [Ref. 8]) assumes an Interlisp users are expert LISP programmers. The system, called HELPSYS, which is usually a integral part of Interlisp system providing online help messages to the user is not implemented for UNIX 4.2. These obstacles result in a steep learning curve to one who desires to use Interlisp without LISP programming experience. Only hindsight can say that the struggle and frustration needed to become productive in this environment were well worth the effort. The impact of seeing these powerful tools in action was an experience that paralleled viewing a rare piece of art that one had only previously read about.

It is incredible to watch the speed with which a database is created by MASTERSCOPE on the RPL system, which consists of 77 LISP functions. The information available through queries to this database provided the basic documentation (that was only amplified slightly) for every function shown in Appendix F.

This feature of Interlisp will be a definite asset to future research. The effects of changing a particular mechanism within the RPL system can be determined by making a few database queries. Figure VII-1 shows how the information was obtained for the documentation listed in Appendix F for a single function and illustrates a few simple queries. By substituting the function name with 'all' in the first query, every function in the database will be 'described'.

Before making specific changes to an existing implementation of an operator or system mechanism some concrete data may be needed to verify perceived problem areas. This performance data is readily available through BREAKDOWN. The next section will illustrate this mechanism and demonstrate the use of the otherwise disastrous backspace key as an RPL interrupt, allowing the programmer to enter LISP commands for debugging, editing and/or performance testing. Note that the message

```
?>  interrupted below READP

(READP broken)
:
```

will occur when the backspace is pressed at the RPL prompt. The ':' prompt is the LISP break prompt and the programmer has the freedom to execute any LISP command. The command

```
: return NIL
```

will restore RPL to the same position where the session was

interrupted. Note that the system will not redisplay the line, therfore the cursor will be on the first column of what appears to be a blank line.

The array reduction operator was implemented in LISP as part of the kernel. The main consideration for the LISP implementation was to make the operator more efficient. The extensional definition suggested by MacLennan [Ref. 2 p. 65] using a 'while' functional was painfully slow. The current implementation takes advantage of the fact that both operands have been evaluated at the time the closure is made (in BIF_APPLY). Therefore, the expression formed as the body of the closure has the operands in evaluated form. As discussed in Chapter 5, this operator could have been easily defined extensionally. In this implementation the operands have to be evaluated in ARRAY_REDUCTION. The results of a performance test using BREAKDOWN is shown in detail in the following section. Of particular note was the minor editing of the function ARRAY_REDUCTION that was done in order to perform the comparison.

This type of analysis can be done for the composition operator and parallel operators. These operators are currently implemented extensionally, and both operators return closures. With the extensional implementation input errors are not detected until the function is applied. Adding 'o' and '||' to the kernel may enhance RPL efficiency considerably.

The design of the RPL system allows the addition of operators to the kernel without a major coding effort. By grouping operators in BIF_APPLY according to the operand(s) requirements, error checking for most operators is already in place. Of course an infix operator being changed from an extensional implementation to the kernel will have to have its extensional definition removed from INTOPS and a representative definition added to SYSYOPS, as well as having its name added to the list BIFTAG_INFIX.

Much work remains to be done to determine which set of operators is best suited for the RPL kernel. This may be answered through a systematic analysis of this prototype with the tools provided by Interlisp. More efficient implementations of some kernel operators is also likely. Additionally, follow on implementations will have more flexibility with RPL notation if a character-at-a-time parser is adopted.

A. USING BREAKDOWN

In order to illustrate the power and flexibility available to do performance analysis, edit functions and create a history of the work performed, the following example was created. This example will use the UNIX function 'script' to record the terminal session. In this session the factorial function will be defined in terms of the RPL array reduction operator. This function will be used as a

```
Script started on Tue Jun 11 21:04:52 1985
 % ilisp
 ISI-INTERLISP 15-MAY-84 ...

 Good evening.

 _load[rpl-int]
 File Created: 8-JUN-85 13:39:37

 RPL-INTCOMS
 expanding LISTP, 65520 used, 2424832 before GC
 /work/mitton/RPL-INT.;2

 _masterscope]
 Masterscope 28-MAR-84... Type HELP<cr> for command summary.

 _. ANALYZE FUNCTIONS ON RECORD
 ..................................................
 expanding LISTP, 131032 used, 2359296 before GC
 .........................done
 _. DESCRIBE EV
 EV[EXP,E]
   calls: NUMBERP,STRINGP,ATOM,MEMBER,LOOKUP,ERROR_HANDLER,
          EV_SPECIAL_CASES,LENGTH,PREFIXOP,INFIXOP
   called by: EXECUTE,DEF_BINDING,DISPLAY,EV_SPECIAL_CASES,
              MAPEV,EVSEQ,INFIXOP,PREFIXOP,RPAPPLY,
              ARRAY_REDUCTION,RPL_REPEAT,MAKE_UNIQUE
   binds:     X,TAG
   uses free: SPECIAL_CASES
 NIL
 _. WHO CALLS ERROR_HANDLER
 (DISPLAY EVRANGE EVSEQ RPAPPLY ARRAY_REDUCTION MIN_SET
 RPL_REPEAT EXECUTE EV EV_SPECIAL_CASES INFIXOP PREFIXOP
 BIF_APPLY ARRAY_CONCATENATION HEAD MAX_SET MEM
 _. WHO USES ERRORCODE
 (RPL ERROR_HANDLER FILTER READ_USER_DEFS DEF_BINDING
 DISPLAY EV_SPECIAL_CASES EVSEQ INFIXOP PREFIXOP RPAPPLY
 BIF_APPLY RPL_REPEAT)
 _. WHO SETS ERRORCODE
 (RPL ERROR_HANDLER FILTER READ_USER_DEFS)
 _.OK
 NIL
 % ^D
 script done on Tue Jun 11 21:23:02 1985
 %
```

Figure VII-1 -- Example of LISP's Masterscope Feature

benchmark to examine the 'red' operator implementation. The current implementation of 'red' is done in LISP using the techniques described in Chapter 5, and will be compared to two extensional implementation.

Amplifying remarks for notes in Figure VII-2:

1. The file 'brkdwn.sess' initialized by the UNIX 'script' function to record the terminal session.

2. RPL system functions are loaded into Interlisp.

3. The command 'BREAKDOWN' followed by a list of functions will internally mark these functions for monitoring in the performance analysis during the session.

4. Factorial function defined as a benchmark.

5. 'Backspace' (BS) key causes an interrupt to the RPL session.

6. The command 'breakdown[]' will zero internal counters for the performance analysis. This is done so that any data accumulated during RPL loading and the definition of 'fac' will not distort analysis.

7. The command 'brkdwnresults[]' is used to verify that the counters are zeroed.

8. The command 'return NIL' is used return to RPL.

9. The RPL command '(fac 5)' is entered for benchmarking.

10. BS interrupt (See #5).

11. The data generated from BREAKDOWN is retrieved.

12. The LISP editor is used to modify ARRAY_REDUCTION. This is necessary since f and i are passed in evaluated form in the current implementation.

13. Return to RPL (See #8).

14. An extensional version of the array reduction operator is defined, and a factorial function using this operator is defined.

15. BS interrupt (See #5).

16. Counters are zeroed using 'breakdown[]' command.

17. Return to RPL and benchmark program ran ('facext').

18. BS interrupt (See #5).

19. Performance data is obtained.

20. Return to RPL (See #8).

21. Array reduction is defined by translating the definition used by Maclennan [Ref. 2]. This illustrates the shift in the use of sequences to lists as functional arguments. The poor preformance shown below led to the implementation used in the first example.

22. BS interrupt (See #5).

23. Counters in BREAKDOWN zeroed.

24. Return to RPL and benchmark program ran (FAC).

25. BS interrupt (See #5).

26. Performance data is obtained.

27. '^C' terminates the Interlisp process and returns the process to UNIX.

28. '^D' terminates the session and writes 'brdwn.sess'.

```
                                                                    NOTE
% script brkdwn.sess                                                 1

Script started on Sat Jun  8 12:46:09 1985
% iIisp
ISI-INTERLISP 15-MAY-84 ...

Hi.
load[rpI-int]                                                       2
File Created: 6-JUN-85 04:04:45
RPL-INTCOMS

expanding LISTP, 65520 used, 2424832 before GC
/work/mitton/RPL-INT
_breakdown (EV EV_SPECIAL_CASES RPAPPLY INFIXOP PREFIXOP BIF_APPLY]  3
(EV EV_SPECIAL_CASES RPAPPLY INFIXOP PREFIXOP BIF_APPLY)
_[RPL]
Loading RPL--- DO YOU WANT TO RESUME A PREVIOUS RPL SESSION? <y/n> N


RPL INTERPRETER ON LINE!!

?> fac n == (((op times) red 1) (Iistrange 1 to n]                  4

?> interrupted below READP                                          5

(READP broken)
:breakdown[]                                                        6
(EV EV_SPECIAL_CASES RPAPPLY INFIXOP PREFIXOP BIF_APPLY)
:brkdwnresuIts[]                                                    7

FUNCTIONS     TIME     # CALLS   PER CALL     %
EV            0.0         0       0.0        0
EV_SPECIAL_CASES
              0.0         0       0.0        0
RPAPPLY       0.0         0       0.0        0
INFIXOP       0.0         0       0.0        0
PREFIXOP      0.0         0       0.0        0
BIF_APPLY     0.0         0       0.0        0
TOTAL         0.0         0       0.0
NIL
:return NIL                                                         8
READP = NIL
(fac 5]                                                             9

120
```

Figure VII-2 -- RPL TerminaI Session Using BREAKDOWN

```
?> interrupted below READP                                          10

(READP broken)
:brkdwnresults[]                                                    11

FUNCTIONS     TIME     # CALLS   PER CALL     %
EV            1.12        64      0.0175     34
EV_SPECIAL_CASES
              0.224       3       0.0746667  7
RPAPPLY       0.24        23      0.0104348  7
INFIXOP       0.592       16      0.037      18
PREFIXOP      0.048       2       0.024      1
BIF_APPLY     1.056       16      0.066      32
TOTAL         3.28        124     0.0264516
NIL
:editf(ARRAY_REDUCTION]                                             12
edit
*F FNC
*F FNC
*0 P
(SETQ FNC (CADDDR EXP))
*3 P
(CADDDR EXP)
*(-1 EV)
*(N EA]
*P
(EV CADDDR EXP EA)
*BI 2 3
*P
(EV (CADDDR EXP) EA)
*F START
*P
... START (CADDDR &))
*0 P
(SETQ START (CADDDR &))
*3 P
(CADDDR (CDDR EXP))
*(-1 EV)
*(N EA]
*P
(EV CADDDR (CDDR EXP) EA)
*BI 2 3
*P
(EV (CADDDR &) EA)
*OK
ARRAY_REDUCTION
:return NIL                                                         13
READP = NIL
```

Figure VII-2 -- RPL Terminal Session Using BREAKDOWN (continued)

```
f redext i == (func ?A (reduce ?A by f from i]                    14

?> facext n == (((op times) redext 1) (listrange 1 to n]

?> interrupted below READP                                        15

(READP broken)
:breakdown[]                                                      16
(EV EV_SPECIAL_CASES RPAPPLY INFIXOP PREFIXOP BIF_APPLY)
:return NIL                                                       17
READP = NIL
(facext 5]

 120

?> interrupted below READP                                        18

(READP broken)
:brkdwnresults[]                                                  19

FUNCTIONS      TIME      # CALLS   PER CALL      %
EV             1.184       67      0.0176716   35
EV_SPECIAL_CASES
               0.224       4       0.056       7
RPAPPLY        0.352       23      0.0153043   11
INFIXOP        0.56        16      0.035       17
PREFIXOP       0.064       2       0.032       2
BIF_APPLY      0.96        15      0.064       29
TOTAL          3.344       127     0.0263307
NIL
:return NIL                                                       20
READP = NIL
s1 == (rsec sel 1]                                                21

?> s2 == (rsec sel 2]

?> p == ((rsec <> empty) o s2]

?> cdr == ((I (\ bar) (un o epsilon)) o s2]

?> arg == (I !! (tl o epsilon]

?> f RED i == (s1 o (((((f o arg) !! cdr) o DELTA) while p) o (lsec i ,]

?> FAC n == (((op times) RED 1) (listrange 1 to n]

?> interrupted below READP                                        22
```

Figure VII-2 -- RPL Terminal Session Using BREAKDOWN (continued)

```
(READP broken)                                                    23
:breakdown[]
(EV EV_SPECIAL_CASES RPAPPLY INFIXOP PREFIXOP BIF_APPLY)
:return NIL                                                       24
READP = NIL
(FAC 5)

 120

?> interrupted below READP                                        25

(READP broken)
:BRKDWNRESULTS[]                                                  26

FUNCTIONS     TIMETIME     # CALLS   PER CALL      %
EV            9.392          497     0.0188974   40
EV_SPECIAL_CASES
              1.808          33      0.0547879   8
RPAPPLY       3.2            161     0.0198758   14
INFIXOP       2.064          51      0.0404706   9
PREFIXOP      2.88           100     0.0288      12
BIF_APPLY     3.872          58      0.0667586   17
TOTAL         23.216         900     0.0257956
NIL
^C                                                                27
% ^D                                                              28
script done on Sat Jun  8 13:03:33 1985
```

Figure VII-2 -- RPL Terminal Session Using BREAKDOWN (continued)

APPENDIX A - ORIGINAL RPL GRAMMAR

$$session = command^{+}\ \textbf{done}$$

$$command = \left\{ \begin{array}{l} prefixid\ [\ identifier\ ] \equiv expression \\ \textbf{display}\ expression \end{array} \right\}$$

$$expression = \left\{ \begin{array}{l} [\ expression\ infix\ ]\ application \\ superscription \end{array} \right\}$$

$$application = \left\{ \begin{array}{l} [\ application\ ]\ primary \\ \textbf{iter}\ [\ primary \rightarrow primary\ ] \end{array} \right\}$$

$$superscription = expression\ \textbf{sup} \left\{ \begin{array}{l} application \\ + \\ * \end{array} \right\}$$

$$primary = \left\{ \begin{array}{l} literal \\ prefixid \\ [\ \left\{ \begin{array}{l} infix \\ infix\ primary \\ primary\ infix \\ primary \rightarrow primary\ ;\ primary \end{array} \right\}\ ] \\ (\ expression\ [\ ..\ expression\ ]\ ) \\ \{\ expression\ [\ ..\ expression\ ]\ \} \\ <\ primary\ ,\ \cdots\ > \\ \textbf{file}\ string \end{array} \right\}$$

$$infix = infixop\ [\ \textbf{bar}\ ]$$

$$identifier = letter \left[\begin{array}{l} letter \\ digit \end{array} \right]^{*}\ prime^{*}$$

$$prime = '$$

$$literal = \left\{ \begin{array}{l} digit^{+}\ [\ .\ digit^{+}\ ] \\ string \\ \textbf{true} \\ \textbf{false} \end{array} \right\}$$

$$string = ``\ char^{*}\ ''$$

infixop =
sel | , : cup member nomem !subset subset = -> <- restr ; cl cr cap \
@ hat ! cat @ . || $ red + - times divide != < > <= >=
andsign orsign cart

prefixid = { *identifier* / *prefixop* }

prefixop =
- un cur unc theta size str DELTA inv dom rng mem Lm Rm Mm run lun bun
init term alpha omega ALPHA OMEGA min max mu index select join as sa sa0
rp rpi rsort sort unimg all ssm img curry uncurry PHI Id while upsilon
phi delta PI extend restrict wig not

APPENDIX B - IMPLEMENTED RPL GRAMMAR

```
session          = (command)* (done)

command          = prefixid [identifier] == expression
                 = identifier infixid identifier == expression
                 = file string == expression
                 = [display | dis | d] expression
                 = val identifier
                 = env [identifier]

expression       = (expression infix expression)
                 = application
                 = superscription

application      = primary
                 = (application primary)
                 = (iter primary -> primary)

superscription   = (expression sup application)
                 = (expression sup +)
                 = (expression sup **)

primary          = literal
                 = prefixid
                 = (op infix)
                 = (rsec infix primary)
                 = (lsec primary infix)
                 = (if primary -> primary ; primary)
                 = (rel (expression : expression)  ... )
                 = (seqrange expression to expression)
                 = (setrange expression to expression)
                 = (listrange expression to expression)
                 = (seq primary  ...  )
                 = (set primary  ...  )
                 = (list primary ...  )
                 = (file string)
                 = (func formals expression)
                 = empty

infix            = infixid
                 = (infixid bar)

formals          = identifier
                 = (identifier+)

identifier       = letter [letter | digit]* prime*

prime            = '

literal          = digit+ [. digit+]
                 = string
                 = true
                 = false

string           = "char*"

prefixid         = identifier
                 = prefixop

infixid          = identifier
                 = infixop
```

```
prefixop        = un
                = cur
                = unc
                = theta          Primitive Extensionals
                = epsilon
                = size

                ************************************************

                = DELTA
                = cnv
                = rev
                = dom
                = rng
                = mem            Non-Primitive Extensionals
                = run                   (Group I)
                = lun
                = bun
                = init
                = term

                = hd
                = tl
                = alpha
                = omega
                = ALPHA
                = OMEGA
                = min            Non-Primitive Extensionals
                = max                   (Group II)
                = uset
                = mu
                = as
                = rsort
                = sort
                = ssm

                ************************************************

                = curry
                = uncurry        Primitive Intensionals
                = I

                ************************************************

                = while
                = upsilon
                = phi            Non_Primitive Intensionals
                = delta
                = PI
                = wig

                ************************************************

                = not            Miscellaneous
```

```
infixop          = sel
                 = !
                 = ,              Primitive Extensionals
                 = :
                 = cup
                 ***********************************************

                 = member
                 = nomem
                 = Lm
                 = Rm
                 = Mm
                 = !subset
                 = subset         Non-Primitive Extensionals
                 = =                      (Group I)
                 = filter
                 = ->
                 = <-
                 = restr

                 = ;
                 = index
                 = select
                 = join
                 = unimg
                 = all
                 = cl
                 = cr
                 = cap            Non-Primitive Extensionals
                 = \                        (Group II)
                 = rp
                 = rpi
                 = #
                 = @hat
                 = xi
                 = !
                 = sa
                 = cat

                 ***********************************************

                 = @
                 = o
                 = !!             Primitive Intensionals
                 = $
                 = img
                 = PHI


                 = red
                 = extend         Non-Primitive Intensionals
                 = restrict

                 ***********************************************

                 = +
                 = -
                 = times
                 = divide ! /
                 = != ! <>
                 = <              Miscellaneous
                 = >
                 = <=
                 = >=
                 = andsign ! and
                 = orsign ! or
                 = cart
```

APPENDIX C - RPL OPERATORS

A. INTRODUCTION

This appendix will describe all the RPL operators implemented to date. Sections B - L each cover one of the operator classes outlined in Chapter VI. Because all of the data input operators are included in the 'Special Operator' class, it is discussed first, followed by the the remaining classes in the order indicated in Chapter VI. Also, to provide easier access to the operators, an index is included at Appendix D.

The format utilized provides the user with the name of the operator in functional terms, its syntax, input(s)/output, a description of what the operator does, and one or more examples. Each example is written as an RPL command which will return a result. Therefore, definition of variables is kept to a minimum to keep the structures visible so the user can follow more easily what is happening.

Long input definitions and output are highly formatted in this appendix. The user must realize that output from the interpreter itself is not as structured. A large relation in RPL is just a LISP list, and so when it is printed to the screen, it is printed as a single long list, modified slightly by RPL routines. Therefore, the output

represented in this Appendix has been nicely formatted to clarify the structures involved and to help the understanding of the user.

Arguments to RPL operators can take various forms, but are all variations of the three basic types - scalars, sets or relations. In general, data types will be represented through the use of lowercase letters as follows:

x, y, z	==>	scalar, or anything
s	==>	set
t, u	==>	relation (table), sequence or list
d	==>	relation - database
f, g	==>	function
p	==>	boolean function
m, n	==>	integers

The operators have generally been classified by the type of argument they apply to, e.g., set, relation, sequence, array. Sequences, arrays, records and the like are all special forms of a relation. Another unique form of relation utilized by several of the higher order operators is the data structure.

A RPL data structure consists of two parts, the form part, R, and the data part, D. These two parts are combined as a RPL list. Thus, the internal structure appears as:

```
(rel (1 D) (2 R))
```

R, the form component, is a relation represented as a sequence of indices to the data elements. These indices can be anything the user desires, as long as they all are distinct. The data part, D, is also a relation which relates the indices to their respective data values. For example, consider a data structure for the sequence,

(10, 20, 30, 40, 50).

For simplicity, let the form part, R, be represented by the sequence, (1, 2, 3, 4). Internally, R would look like:

(rel (1 2) (2 3) (3 4))

This would lead to the data part, D, with an internal form:

(rel (1 10) (2 20) (3 30) (4 40))

Together these components would produce the data structure:

```
S = (rel (1 (rel (1 10) (2 20) (3 30) (4 40)))
         (2 (rel (1 2) (2 3) (3 4))) )
```

In this appendix, a data structure will be represented by the capital letter, 'S'. This letter is used to distinguish it from the lowercase letters which are used to represent other argument/data types in the language.

For additional and developmental information concerning any of the operators in this Appendix, see MacLennan [Ref. 2]. Some operators have been altered, added or deleted from the original set proposed by MacLennan. Appendix E summarizes in tabular form, the evolution from the original proposal to the implemented version of operators. It provides a quick reference to the syntax of

the operators in their input form and contrasts this input form with the publication form created through the use of the Unix 'eqn' package.

B. SPECIAL OPERATORS

1. Relation Definition

a. Syntax: (rel (x1 : y1) (x2 : y2) ...)

b. Input(s): anything
Output: relation

c. Description: The 'rel' operator is the general mechanism to create a relation in RPL. It normally uses the pair-making operation described in the next section to convert the data given into the internal representation for a relation.

d. Example(s):

```
?> (rel (1 : 2) (3 : 4) (4 : 5)]
 (rel (1 2) (3 4) (4 5))
```

2. Set Definition

a. Syntax: (set x1 x2 x3 ...)

b. Input(s): anything
Output: relation (set)

c. Description: The 'set' operator evaluates and transforms the data items given into the internal representation for an RPL set.

d. Example(s): Suppose a = 3 and b = 5:

```
?> (set 1 2 a 4 b]
 (set 1 2 3 4 5)
```

3. Sequence Definition

a. Syntax: (seq x1 x2 x3 ...)

b. Input(s): anything
Output: relation

c. Description: The 'seq' operator is an easier way to enter a special kind of relation called a sequence. It is up to the user to insure that the data item he is creating is a pure sequence, i.e., has no redundant elements in it. This mechanism can also be used to enter certain types of directed graphs when redundant elements are included.

d. Example(s):

```
(1)  ?> (seq 1 2 3 4 5]
        (rel (1 2) (2 3) (3 4) (4 5))

(2)  ?> (seq 8 3 7 7 5 4]
        (rel (8 3) (3 7) (7 7) (7 5) (5 4))
```

4. List Definition

a. Syntax: (list x1 x2 x3 ...)

b. Input(s): anything
Output: relation

c. Description: The 'list' operator is an easier method to enter a relation which looks like an array. It sets up an internal structure which orders the data given by relating an index, starting with 1, to the value provided. It is called a list after its primary use, for making argument lists for infix functions.

d. Example(s): Suppose x = 30:

```
?> (list 10 20 x 40]
 (rel (1 10) (2 20) (3 30) (4 40))
```

5. Range Definition - Set, Sequence, and List.

a. Syntax: (setrange m to n)
(seqrange m to n)
(listrange m to n)

b. Input(s): integers
Output: relation

c. Description: These operators are used to easily create relatively large numeric relations. The values within the range from m to n are transformed into the appropiate structure.

d. Example(s):

```
(1)  ?> (setrange 2 to 5]
       (set 2 3 4 5)

(2)  ?> (seqrange 1 to 5]
       (rel (1 2) (2 3) (3 4) (4 5))

(3)  ?> (listrange 10 to 30]
       (rel (1 10) (2 20) (3 30))
```

6. Direct Function Definition

a. Syntax: name == (func (arg) (body))

b. Input(s): argument list; body of definition
Output: RPL function

c. Description: The syntax includes the entire command line required to execute a 'func'. The function components provided are converted into the RPL internal function representation and the environment of definition is attached. However, this environment is never displayed to the screen in evaluated form. The 'env' command will allow the user to see the environment of any function in its definitional form. The 'val' command will allow the user to see the internal representation of a function, but the environment will not be displayed.

d. Example(s):

```
?> sum == (func (x y) (x + y)]
?> val sum
 (closure (x y) (x + y))
?> (sum (list 2 3)]
 5
```

7. Infix to Prefix Conversion

a. Syntax: (op f)

b. Input(s): infix function
Output: prefix function

c. Description: The 'op' operator transforms an infix operator into a prefix operator so that it can be composed with other functions. Once converted the arguments to this function must be provided in the form of a binary list.

d. Example(s):

```
?> ((op +) (list 2 3]
 5
```

8. Left Section and Right Section

a. Syntax: (lsec x f)
(rsec f x)

b. Input(s): x, anything; f, infix operator
Output: function

c. Description: These two operators allow the user to fix either the left or right argument to an infix function. Thus x must be a suitable argument to the infix function provided.

d. Example(s):

(1)
```
?> ((lsec 3 +) 2]
 5
```

(2)
```
?> ((rsec = 3) 2]
 false
```

9. Conditional Functional

a. Syntax: (if p -> f ; g)

b. Input(s): p, predicate - boolean function
f, g - any function
Output: function

c. Description: This functional creates a function which when given an argument will pass it to the predicate. If true, then f will be applied to the argument, else g will be applied to the argument.

d. Example(s): Suppose the user wanted to add or subtract two numbers based on the sign of the first number. The following predicate and functions could be used (See Chapter VI for explaination of function definitional forms:

```
?> p x == ((x sel 1) < 0]
?> f == (op +]
?> g == (op -]
?> ((if p -> f ; g) (list 3 2]
 1
```

10. Iteration Functional

a. Syntax: (iter p ; f)

b. Input(s): p, predicate (boolean function)
f, any function
Output: anything

c. Description: This functional produces a function which when given an argument will apply f to that argument at least once. Then if the predicate applied to the result of the first application of f is true, it will apply f to the result. This cycle continues until the predicate fails.

d. Example(s): Consider a trivial case where the user wanted the argument to be doubled until it was greater than 50, and then return the result:

```
?> p == (rsec <= 50]
?> f = (rsec times 2]
?> ((iter p -> f) 4]
 64
```

11. Superscription

a. Syntax:
```
(t sup +)
(t sup **)
(t sup -1)
(f sup n)
```

b. Input(s): t, relation
f, function; n, positive integer
Output: relation; function

c. Description: This operator has four cases as shown above and is the only one that can be applied to both extensional relations and functions. When the right argument is '+' a transitive closure is performed. When a '**' is provided, a reflexive transitive closure is done. Note, a double asterisk is required because of a conflict with the use of the '*' symbol in LISP. When a '-1' is the right argument, the converse of t is returned. When the left argument is a function and the right argument is a positive integer, the function is composed with itself n times.

d. Example(s): Let t = (seq 1 2 3 4)
f = (x + 2)

```
(1)  ?> (t sup +]
       (rel (1 2) (2 3) (3 4) (1 3) (2 4) (1 4))

(2)  ?> (t sup **]
       (rel (1 1) (2 2) (3 3) (4 4) (1 2)
            (2 3) (3 4) (1 3) (2 4) (1 4))

(3)  ?> (t sup -1]
       (rel (2 1) (3 2) (4 3))

(4)  ?> ((f sup 2) 2]
       8
```

12. Formalization Functional

a. Syntax:
```
(f (+ bar) g),
(f (- bar) g),
(f (times bar g), . . .
```

b. Input(s): infix operator; functions
Output: function

c. Description: The 'bar' operator converts any infix operator into a functional which takes two functions as arguments. The resulting functional will apply the input functions f and g to an appropiate argument and then apply the 'barred' infix operator to the results.

d. Example(s): Consider a definition for a function which squares its arguments. It utilizes the Identity function, I, which is explained in the next section:

```
?> sqr == (I (times bar) I]
?> (sqr 4]
 16
```

13. Empty Set or Relation

a. Syntax: empty

b. Input(s): none
Output: set or relation

c. Description: This operator is actually a data element which represents the empty set or relation. It is normally used to initialize sets or relations and may be returned as the result of other operations.

d. Example(s):

```
?> x == empty
```

C. GLOBAL OPERATORS

1. Equality and Inequality

a. Syntax: (x = y)
(x != y) or (x <> y)

b. Input(s): anything
Output: boolean

c. Description: Compares any two RPL data types based upon their mathematical equivalence, not form.

d. Example(s):

```
(1)  ?> (2 = 3]          (2)  ?> (2 != 3]
      false                     true

(3)  ?> ((set (1 : 2) (2 : 3) (3 : 4)) = (seq 1 2 3 4]
      true
```

2. Duplication

a. Syntax: (DELTA x)

b. Input(s): anything
   Output: relation

c. Description: Duplicates the argument and returns a relation in the form of a binary list.

d. Example(s):
```
?> (DELTA "a"]
 (rel (1 "a") (2 "a"))
```

3. Identity

a. Syntax: (I x)

b. Input(s): anything
   Output: anything

c. Description: Returns the input unchanged.

d. Example(s):
```
?> (I 3]
 3
```

4. Pair Formation

a. Syntax: (x : y)

b. Input(s): anything
   Output: elementary pair

c. Description: Used to create the elements of a relation in conjunction with other operators. It has no meaning by itself.

d. Example(s): (1 : 2) ==> (1 2)

5. Head, Tail

a. Syntax: (hd z)
(tl z)

b. Input(s): elementary pair
Output: anything

c. Description: Given a LISP elementary pair, i.e., a dotted pair, 'hd' will return the first element, 'tl' will return the last element. These operations are used within function definitions to extract pieces of a relation which can be further processed.

d. Example(s):

```
(1)  ?> (hd (10 : 20]
          10

(2)  ?> (tl (10 : (rel (3 : 4) (4 : 5]
          (rel (3 4) (4 5))
```

6. Pair List

a. Syntax: (x , y)

b. Input(s): anything
Output: relation

c. Description: Converts the two inputs into the relational form of a binary list.

d. Example(s):
```
?> (20 , 30]
 (rel (1 20) (2 30))
```

7. Unit Set

a. Syntax: (un x)

b. Input(s): anything
Output: set

c. Description: Converts the input data item to a set containing that single data item.

d. Example(s):
```
?> (un "dog"]
 (set dog)
```

D. ARITHMETIC OPERATORS

1. Sum, Difference, Product, Quotient

a. Syntax:
```
(x + y)
(x - y)
(x times y)
(x divide y)  or  (x / y)
```

b. Input(s): numeric, real or integer
Output: numeric, real or integer

c. Description: Normal mathematical operations. If either input is a real, the result will be a real, except in division. If the numerator is integer, an integer division will be executed.

d. Example(s):

```
(1)  ?> (2 + 3]          (2)  ?> (3.125 - 2]
       5                        1.125

(3)  ?> (3 - 2]          (4)  ?> (2 * 4]
       1                        8

(5)  ?> (2 divide 4]     (6)  ?> (2.0 / 4]
       0                        0.5
```

2. Less, Greater, Less or Equal, Greater or Equal

a. Syntax:
```
(x < y)
(x > y)
(x <= y)
(x >= y)
```

b. Input(s): numeric, real or integer
Output: boolean

c. Description: Conventional relational operators.

d. Example(s):

```
(1)  ?> (2 < 3]          (2)  ?> (2 >= 3]
       true                     false
```

D. LOGICAL OPERATORS

1. Conjunction, Disjunction, Negation

a. Syntax: (x andsign y) or (x and y)
(x orsign y) or (x or y)
(not x)

b. Input(s): boolean(s)
Output: boolean

c. Description: Conventional logical operators.

d. Example(s):

(1) ?> (true andsign true]
true

(2) ?> ((2 < 3) or (2 > 3]
true

(3) ?> (not (3 = 3]
false

E. SET OPERATORS

1. Maximum, Minimum

a. Syntax: (max s)
(min s)

b. Input(s): numeric set
Output: number

c. Description: Returns the maximum or minimum element of the input set, respectively.

d. Example(s):

(1) ?> (max (set 4 8 2 10 9]
10

(2) ?> (min (set 4 8 2 10 9]
2

2. Relational Sort, Sort

a. Syntax: (rsort s)
(sort s)

b. Input(s): numeric set
Output: relation

c. Description: The input set is sorted in ascending order and converted into a sequence for rsort, and a list for sort.

d. Example(s):

(1) ?> (rsort (set 4 8 2 10 9]
(rel (2 4) (4 8) (8 9) (9 10))

(2) ?> (sort (set 4 8 2 10 9]
(rel (1 2) (2 4) (3 8) (4 9) (5 10))

3. Element Selection

a. Syntax: (epsilon r)

b. Input(s): set or relation
Output: anything

c. Description: Returns the first element of the input provided.

d. Example(s):

(1) ?> (epsilon (set 4 8 2 10 9]
4

(2) ?> (epsilon (rel (1 : 2) (2 : 3]
(1 2)

4. Unique Element Selection

a. Syntax: (theta s)

b. Input(s): unit set
Output: anything

c. Description: Extracts the single member of a unit set and returns it.

d. Example(s): ?> (theta (set "dog"]
dog

5. Unique Set

a. Syntax: (uset r)

b. Input(s): set or relation
Output: set or relation

c. Description: Eliminates redundant elements from the input structure provided.

d. Example(s):

```
?> (uset (set 4 8 2 4 10 9]
 (set 4 8 2 10 9)
```

6. Intersection, Union and Set Difference

a. Syntax:

```
(s cap r)
(s cup r)
 (s \ r)
```

b. Input(s): set or relation
Output: set or relation

c. Description: Conventional set operations.

d. Example(s):

(1)
```
?> ((set 1 2 3) cap (set 2 3 4]
 (set 2 3)
```

(2)
```
?> ((set 1 2 3) cup (rel (1 : 2) (2 : 3]
 (set 1 2 3 (1 2) (2 3))
```

(3)
```
?> ((set 1 2 3) \ (set 2 3 4]
 (set 1)
```

(4)
```
?> ((set 1 2 3) \ (set 1 2 3]
 empty
```

7. Cartesian Product

a. Syntax: (s cart r)

b. Input(s): set or relation
Output: relation

c. Description: None required.

d. Example(s):

```
?> ((set 1 2) cart (set 5 6]
 (rel (1 5) (1 6) (2 5) (2 6))
```

8. Cardinality

a. Syntax: (size r)

b. Input(s): set or relation
Output: integer

c. Description: Returns the number of elements in the input set or relation.

d. Example(s):

(1) ?> (size (set 4 8 2 10 9]
5

(2) ?> (size (rel (1 : 3) (3 : 5) (5 : 7]
3

9. Membership, Nonmembership

a. Syntax: (x member r)
(x nomem r)

b. Input(s): anything; set or relation
Output: boolean

c. Description: Verifies if x is or is not a member of the input set or relation.

d. Example(s):

(1) ?> (2 member (set 1 2 3]
true

(2) ?> ((1 : 2) nomem (rel (1 : 2) (2 : 3)]
false

10. Improper Subset, Proper Subset

a. Syntax: (s !subset r)
(s subset r)

b. Input(s): set or relation
Output: boolean

c. Description: Verifies that all members of s are members of r. The cardinality of s must be less than the cardinality of r for a proper subset.

d. Example(s):

```
(1)  ?> ((set 1 2 3) !subset (set 1 2 3)]
      true

(2)  ?> ((set 1 2 3) subset (set 1 2 3]
      false

(3)  ?> (rel (1 : 2)) subset (set 4 (1 : 2) 5]
      true
```

F. RELATION OPERATORS

1. Selection

a. Syntax: (t sel x)

b. Input(s): anything; relation
Output: anything

c. Description: Given the left member of a relation, x, the associated right member of the first occurence of x in t will be returned.

d. Example(s):

```
?> t == (rel (1 : 2) (2 : 3) (1 : 3) (2 : 4)]
?> (2 sel ]
 3
```

2. Construction

a. Syntax: (t # u)

b. Input(s): relations
Output: relation

c. Description: Constructs a table (relation) which relates each common left member of t and u, to a list created by selecting the respective right members from t and u by using the common left member as a target. When creating the list, the right member associated with the first occurence of the target is used.

d. Example(s):

```
?> t == (rel (1 : 2) (2 : 3) (1 : 3) (2 : 4)]
?> u == (rel (1 : 8) (2 : 9) (3 : 10)]
?> (t # u]
 (rel (1 (rel (1 2) (2 8)))
      (2 (rel (1 3) (2 9))))
```

3. Converse

a. Syntax: (cnv t) or (t sup -1)

b. Input(s): relation
Output: relation

c. Description: Returns a table where each element of table t has the left and right member inverted. See special operator section for other uses of the 'sup' syntax.

d. Example(s):

```
?> t == (rel (1 : 2) (2 : 3) (1 : 3) (2 : 4)]
?> (cnv t]
 (rel (2 1) (3 2) (3 1) (4 2))
```

4. Extensional Curry, Extensional Uncurry

a. Syntax: (cur t)
(unc t)

b. Input(s): relation
Output: relation

c. Description: Given an extensional representation of an infix function in either curried form or uncurried form, these operators will convert one form to the other. Each element in the uncurried form of such a table consists of the function argument list paired with the result of applying the function to these arguments. In curried form, the resulting table is the equivalent of fixing the left member of the infix operator. This left member is paired with another table which contains all potential right members paired to

the result of applying the function to the fixed left member.

d. Example(s): Consider a portion of an uncurried table which represents the '+' function:

```
?> t == (rel ((1 , 1) : 2) ((1 , 2) : 3)
             ((1 , 3) : 4) ((2 , 1) : 3)
             ((2 , 2) : 4) ((2 , 3) : 5) ]
?> (cur t]
 (rel (1 (rel (1 2) (2 3) (3 4)))
      (2 (rel (1 3) (2 4) (3 5))) )
```

5. Ordered Union

a. Syntax: (t ; u)

b. Input(s): relations
Output: relation

c. Description: Creates a table where all elements of t are added to u, replacing any corresponding elements already there.

d. Example(s):

```
?> t == (rel (1 : 2) (2 : 3) (3 : 4)]
?> u == (rel (2 : 4) (3 : 6) (4 : 7)]
?> (t ; u]
 (rel (1 2) (2 3) (3 4) (4 7))
```

6. Primitive Relative Product

a. Syntax: (t ! u)

b. Input(s): relations
Output: relation

c. Description: For an element in t, its right member is used as a target in u, producing a set of values associated with the target. New elements for the resulting table are created by pairing the left member of the element in t with each value in this set. The resulting table contains all the elements created by the above process for each element in t.

d. Example(s):

```
?> t == (rel (1 : 2) (2 : 3)]
?> u == (rel (2 : 4) (2 : 8) (3 : 6) (3 : 12)]
?> (t ! u]
 (rel (1 4) (1 8) (2 6) (2 12))
```

7. All, Unit Image

a. Syntax: (y all t)
(t unimg x)

b. Input(s): anything; relation (all)
relation; anything (unimg)
Output: set

c. Description: 'all' returns a set of all left members related to the target right member, y. Likewise, 'unimg' returns a set of all right members related to the target left member, x.

d. Example(s):

Let t = (rel (1 : 2) (2 : 3) (1 : 3) (2 : 4))

(1)
```
?> (3 all t]
 (set 2 1)
```

(2)
```
?> (t unimg 2]
 (set 3 4)
```

8. Domain, Range

a. Syntax: (dom t)
(rng t)

b. Input(s): relation
Output: set

c. Description: 'dom' returns all left members of the relation t, and 'rng' returns all right members of t. Neither of these operators eliminate redundant elements.

d. Example(s): Consider the relation shown graphically in Figure C-1 and its input form here:

```
t == (rel (1 : 2) (2 : 4) (2 : 5) (3 : 5) (5 : 5)
          (5 : 6) (7 : 6) (8 : 7) (8 : 8) (9 : 7) )
```

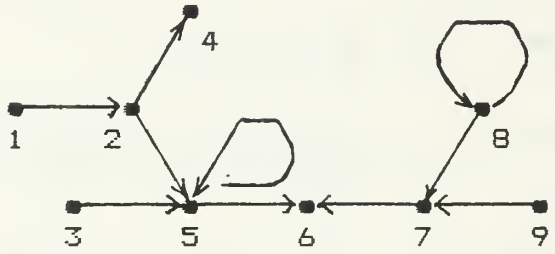


Figure C-1 Arrow Diagram for Relation t

```
(1)  ?> (dom t]
        (set 1 2 2 3 5 5 7 8 8 9)

(2)  ?> (rng t]
        (set 2 4 5 5 5 6 6 7 8 7)
```

9. Initial Members, Terminal Members

a. Syntax: (init t)
(term t)

b. Input(s): relation
Output: set

c. Description: Given a table which represents some relation, the initial members are those which are left members of the relation, but not right members. Conversely, the terminal members are those which are right members, but not left members of the relation. 'init' returns the intial members of a relation, and 'term' returns the terminal members.

d. Example(s): Using the relation in Figure C-1,

```
?> (init t]
   (set 1 3 9)

?> (term t]
   (set 4 6)
```

10. Members

a. Syntax: (mem t)

b. Input(s): relation
Output: set

c. Description: Returns a set of all left and right members of the relation t. Because this operator is defined in terms of the domain, range and union opertors, redundant elements may be left in. The union between the domain and range of t will leave any redundant elements in the range in the result. See reference ## for more information on how LISP implements union.

d. Example(s): Using the relation from Figure C-1,

```
?> (mem t]
 (set 1 3 9 2 4 5 5 5 6 6 7 8 7)
```

11. Left Member, Right Member, Member

a. Syntax: (x Lm t)
(x Rm t)
(x Mm t)

b. Input(s): anything; relation
Output: boolean

c. Description: Verifies if x is a left, right, or either a left or right member of t, respectively.

d. Example(s):

Let t = (rel (1 : 2) (3 : 4) (5 : 6))

```
(1)  ?> (3 Lm t]
       true

(2)  ?> (8 Mm t]
       false

(3)  ?> (5 Rm t]
       false

(4)  ?> (5 Mm t]
       true
```

12. Left Univalent, Right Univalent, Bi-univalent

a. Syntax: (lun t)
(run t)
(bun t)

b. Input(s): relation
Output: boolean

c. Description: A left univalent relation is one in which each element in the domain is unique. In other words, no two different right members can have the same left member. Likewise, a run univalent relation is one in which every right member is unique. Therefore, it follows that a bi-univalent relation, also known as a isomorphism is one that has both unique left and right members. These operators determine if the relation is what is requested.

d. Example(s):

(1) ?> (lun (rel (1 : 2) (2 : 3) (1 : 5)]
false

(2) ?> (run (rel (1 : 2) (2 : 3) (1 : 5)]
true

(3) ?> (bun (rel (1 : 2) (2 : 3) (3 : 4)]
true

G. SEQUENCE OPERATORS

1. First Member, Initial Sequence

a. Syntax: (alpha t)
(ALPHA t)

b. Input(s): sequence
Output: anything; sequence

c. Description: 'alpha' returns the first element of the sequence s, while 'ALPHA' returns the entire sequence except the last element.

d. Example(s):

(1) ?> (alpha (seq 1 2 3 4 5]
1

(2) ?> (ALPHA (seq 1 2 3 4 5]
(rel (1 2) (2 3) (3 4))

2. Last Member, Final Sequence

a. Syntax: t (omega t)
(OMEGA t)

b. Input(s): sequence
Output: anything; sequence

c. Description: 'omega' returns the last element in the sequence t, while 'OMEGA' returns the entire sequence except the first element.

d. Example(s):

(1) ?> (omega (seq 1 2 3 4 5]
5

(2) ?> (OMEGA (seq 1 2 3 4 5]
(rel (2 3) (3 4) (4 5))

3. Cons Left, Cons Right

a. Syntax: (x cl t)
(t cr x)

b. Input(s): x = anything; t = sequence
Output: sequence

c. Description: Any data item is added to the beginning (left) or to the end (right) of the sequence t.

d. Example(s):

(1) ?> (1 cl (seq 2 3 4 5]
(rel (1 2) (2 3) (3 4) (4 5))

(2) ?> ((seq 1 2 3 4) cr 5]
(rel (1 2) (2 3) (3 4) (4 5))

4. Minimize Sequence

a. Syntax: (mu t)

b. Input(s): relation
Output: sequence

c. Description: This operator eliminates redundant edges from a relation which has as its underlying structure a sequence. This type of structure can be obtained as a result of some of the higher order operators discussed in sections K and L. Care must be exercised. If t does not originate from a true sequence, the computation may not halt.

d. Example(s):

```
?> t == (rel (3 : 4) (3 : 6) (3 : 7) (3 : 2)
             (4 : 6) (4 : 7) (4 : 2) (6 : 7)
             (6 : 2) (7 : 2)]
?> (mu t]
 (rel (3 4) (4 6) (6 7) (7 2))
```

5. Sequence of Sequences to Matrix

a. Syntax: (ssm t)

b. Input(s): relation
Output: relation

c. Description: Given a relation in the form of a sequence of sequences, this operator converts it into a relation which represents a matrix. The left member is a list of the column and row number, and the right member is the value at that position.

d. Example(s):

```
?> t == (seq (seq 10 20 30)
             (seq 40 50 60)
             (seq 70 80 90)]
```

```
?> (ssm t]
 (rel
  ((rel (1 1) (2 1)) 10)
  ((rel (1 1) (2 2)) 20)
  ((rel (1 1) (2 3)) 30)
  ((rel (1 2) (2 1)) 40)
  ((rel (1 2) (2 2)) 50)
  ((rel (1 2) (2 3)) 60)
  ((rel (1 3) (2 1)) 70)
  ((rel (1 3) (2 2)) 80)
  ((rel (1 3) (2 3)) 90))
```

6. Sequence to Array

a. Syntax: (t sa n)

b. Input(s): sequence; positive integer
Output: relation (array)

c. Description: Converts the sequence t into an array indexed starting with n.

d. Example(s):

```
?> ((seq 10 20 30) sa 4]
 (rel (4 10) (5 20) (6 30))
```

H. ARRAY OPERATORS

1. Array to Sequence

a. Syntax: (as t)

b. Input(s): relation (array)
Output: relation (sequence)

c. Description: Converts the values of the given array into a sequence.

d. Example(s):

```
?> (as (rel (1 : 10) (2 : 20) (3 : 30) (4 : 40)]
 (rel (10 20) (20 30) (30 40))
```

2. Array Concatenation.

a. Syntax: (t cat u)

b. Input(s): relations (arrays)
Output: relation (array)

c. Description: Concatenates u to t by altering the indices of u to be consecutive with the indices of t.

d. Example(s):

```
?> t == (rel (1 : 10) (2 : 30) (3 : 30)]
?> u == (rel (1 : 40) (2 : 50) (3 : 60)]
?> (t cat u]
 (rel (1 10) (2 20) (3 30) (4 40) (5 50) (6 60))
```

3. Reverse Array

a. Syntax: (rev t)

b. Input(s): relation (array)
Output: relation (array)

c. Description: Returns an array with the values reversed.

d. Example(s): Using t from the example above,

```
?> (rev t]
 (rel (1 30) (2 20) (3 10))
```

I. DATABASE OPERATORS

1. Database Index

a. Syntax: (x index d)

b. Input(s): x, anything (field name)
d, relation (database)
Output: relation

c. Description: Returns a relation which pairs the value associated with field name x, to the entire record that the field name was found in, for all records in d.

d. Example(s): Consider the following database:

```
db1 = (set
       (rel ("#" : 100) ("name" : "Brown")  ("hours" : 10))
       (rel ("#" : 101) ("name" : "Mitton") ("hours" : 8))
       (rel ("#" : 102) ("name" : "Benson") ("hours" : 16))
       (rel ("#" : 103) ("name" : "Murnan") ("hours" : 10))
       (rel ("#" : 104) ("name" : "Garcia") ("hours" : 12)))

      ?> ("hours" index db1]
       (set (10 (rel (# 100) (name Brown) (hours 10)))
            (8  (rel (# 101) (name Mitton) (hours  8)))
            (16 (rel (# 102) (name Benson) (hours 16)))
            (10 (rel (# 103) (name Murnan) (hours 10)))
            (12 (rel (# 104) (name Garcia) (hours 12))))
```

2. Database Select

a. Syntax: (x select d)

b. Input(s): x, anything (field name)
d, relation (database)
Output: function

c. Description: Returns a function which when given a predicate selects those records for which the predicate is true and returns a relation with those records.

d. Example(s): Suppose the user wanted all records which have an 'hours' field equal to 10 from the database, db1, used above. Thus the argument to the functional created by the 'select' operator would be the predicate, (rsec = 10). This predicate compares the value of the x field with the value 10. If true, the record is included in the resulting set.

```
?> (("hours" select db1) (rsec = 10]
 (set (rel (# 100) (name Brown) (hours 10))
      (rel (# 103) (name Murnan) (hours 10))).
```

3. Database Join

a. Syntax: (x join dl)

b. Input(s): x, anything (field name)
dl, relation (database list)
Output: relation (database)

c. Description: This operator performs a natural join on two databases, combining all the fields of both databases, based on the equality of the values in the field specified - x.

d. Example(s): Consider the database, db1, in the 'index' example and the additional database, db2, given below:

```
db2 = (set
       (rel ("#" : 100) ("age" : 32) ("office" : "D3"))
       (rel ("#" : 101) ("age" : 27) ("office" : "A4"))
       (rel ("#" : 102) ("age" : 21) ("office" : "C1"))
       (rel ("#" : 103) ("age" : 45) ("office" : "A2"))
       (rel ("#" : 104) ("age" : 37) ("office" : "B8")))
```

```
?> ("#" join (list db1 db2)]
 (set
 (rel (name Garcia) (hours 12) (# 104) (age 37) (office B8))
 (rel (name Murnan) (hours 10) (# 103) (age 45) (office A2))
 (rel (name Benson) (hours 16) (# 102) (age 21) (office C1))
 (rel (name Mitton) (hours 8)  (# 101) (age 27) (office A4))
 (rel (name Brown)  (hours 10) (# 100) (age 32) (office D3)))
```

K. HIGHER ORDER OPERATORS - RETURNING FUNCTIONS

1. Array Reduction.

a. Syntax: (f red x)

b. Input(s): function; anything
Output: function

c. Description: Given a function f, which will operate on the data of an array, and a starting point, x, this operator produces a function which reduces an array. When executed, the result is set to the starting point, x. f is applied to the result and the first element of

data in the array, producing a new result. f continues to be applied as above until all data elements have been utilized as input. The result is then returned.

d. Example(s): Consider the definition for factorial:

```
?> fac x == (((op times) red 1) (listrange 1 to x]
?> (fac 8]
 40320
```

2. Composition

a. Syntax: (f o g)

b. Input(s): functions
Output: function

c. Description: Produces a function which when given an appropriate argument will apply f to the result of applying g to that argument.

d. Example(s): Consider another definition for the squaring function:

```
?> sqr == (times o DELTA)
?> (sqr 4)
 16
```

3. Curry and Uncurry

a. Syntax: (curry f)
(uncurry f)

b. Input(s): function
Output: function

c. Description: These two operators are used to convert between the two types of infix functions. An infix function which takes a single argument in the form of a list is in uncurried form. When such a function is curried, it produces a functional, which will produce another function when given one of the two arguments that are normally required. This resultant

function fixes this argument and creates a function which takes any other valid argument and returns the same result as if the uncurried version had been given both arguments.

d. Example(s):

```
?> sum == (op +]                    {* uncurried form *}
?> add == (curry sum]
?> f == (add 3]
?> (sum (list 3 5]
 8
?> (f 5]
 8
```

4. Extension

a. Syntax: (t extend f)

b. Input(s): relation; function
Output: functional

c. Description: Produces a functional which when given an argument first checks to see if it is the domain of t. If so, its right member is returned, else the function f is applied to the argument.

d. Example(s): Suppose the user wanted to work with a subrange of the positive integers, say 1 to 50, so that the successor of the argument would be returned if the argument was in this subrange, and an error message would be returned if it was not:

(1)
```
?> t == (seqrange 1 to 50]
?> f x == "Error - not within range"
?> subrange == (t extend f]
?> (subrange 25]
 26
```

(2)
```
?> (subrange 55]
 Error - not within range
```

5. Negation of a Function

a. Syntax: (wig p)

b. Input(s): boolean function
Output: function

c. Description: Returns a function which negates the result of the input boolean function.

d. Example(s): Consider a function to determine if a numeric argument is within the subrange 10 to 20, and then the opposite, a function to determine if the argument is outside the range:

```
?> in-range x == ((x >= 10) and (x <= 20]
?> out-of-range == (wig in-range]
?> (out-of-range 25]
 true
```

6. Paralleling of Functions

a. Syntax: (f !! g)

b. Input(s): functions
Output: function

c. Description: Produces a function from the two input functions which when given an argument list, returns a list of the results of applying f to the first member of the argument list and g to the last member of the argument list.

d. Example(s): Consider a different approach to the in-range function from the last example:

```
?> blist == (((rsec >= 10) !! (rsec <= 20)) o DELTA]
?> (blist 15]
?> (rel (1 true) (2 true))
?> in-range == (and o blist]
?> (in-range 15]
 true
```

7. While Loop

a. Syntax: (f while p)

b. Input(s): function; boolean function
Output: function

c. Description: Produces a function which when given an argument will first test the predicate with the argument. If the predicate succeeds then f is applied to the argument. The result of this application is passed to the predicate and if the predicate again succeeds, f is applied to this result. This cycle continues until the predicate fails. If the predicate fails on the first attempt, the original argument is returned.

d. Example(s): Consider a definition for modulo arithmetic:

```
?> modaux x == ((rsec - x) while ((rsec >= 0) o (rsec - x)]
?> mod == ((uncurry modaux) o rev]
?> (10 mod 4]
 2
```

8. Value of a Node, Data Structure

a. Syntax: (upsilon f)

b. Input(s): function
Output: function

c. Description: Creates a function which takes a data structure and returns the value of the node selected by f.

d. Example(s): Suppose the user wanted a function which would return the value of the first node of a given data structure. Consider a RPL data structure for a sequence:

```
?> S == (list (list 3 4 -2 6 7 -1 2 -4)
              (seqrange 1 to 8]
?> val S
 (rel (1 (rel (1 3) (2 4) (3 -2) (4 6) (5 7)
              (6 -1) (7 2) (8 -4))
      (2 (rel (1 2) (2 3) (3 4) (4 5) (5 6)
              (6 7) (7 8)) )
?> first == (upsilon alpha]
?> (first S]
 3
```

9. Operate on Data, Data Structure

a. Syntax: (delta f)

b. Input(s): function
Output: function

c. Description: Creates a function which will operate on the data part of the RPL data structure. Therefore the function f must accept as a valid argument the relation which represents the data part of the data structure. The resulting function takes a data structure as an argument, applies f to the data component, and returns the modified data structure.

d. Example(s): Suppose the user wanted to add 1 to every data element of the data structure used in the last example:

```
?> f == (lsec (hd (: bar) ((rsec + 1) o tl)) img]
?> add1 == (delta f]
?> (add1 S]
 (rel (1 (rel (1 4) (2 5) (3 -1) (4 7) (5 8)
              (6 0) (7 3) (8 -3))
      (2 (rel (1 2) (2 3) (3 4) (4 5) (5 6)
              (6 7) (7 8)) )
```

10. Operate on Form, Data Structure

a. Syntax: (phi f)

b. Input(s): function
Output: function

c. Description: Creates a function which will operate on the form part of the RPL data structure. Therefore the function f must accept as a valid argument the relation which represents the form part of the data structure. The resulting function takes a data structure as an argument, applies f to the form component, and returns the modified data structure.

d. Example(s): Using the data structure defined preiously, consider a function which will eliminate the first node of the data structure:

```
?> rest == (phi OMEGA]
?> (rest S]
 (rel (1 (rel (1 3) (2 4) (3 -2) (4 6) (5 7)
              (6 -1) (7 2) (8 -4))
      (2 (rel (2 3) (3 4) (4 5) (5 6)
              (6 7) (7 8)) )
```

11. Image of a Data Structure

a. Syntax: (PI f)

b. Input(s): function
Output: function

c. Description: Creates a function, that when given a data structure, applies f to all values in the data part of the structure and returns the modified data structure.

d. Example(s): Now, to add 1 to every value as done in the 'delta' example, the user simply writes:

```
?> add1 == (PI (rsec + 1)]
?> (add1 S]
 (rel (1 (rel (1 4) (2 5) (3 -1) (4 7) (5 8)
              (6 0) (7 3) (8 -3))
      (2 (rel (1 2) (2 3) (3 4) (4 5) (5 6)
              (6 7) (7 8)) )
```

L. HIGHER ORDER OPERATORS - RETURNING DATA

1. Filtering Sequences

a. Syntax: (p xi t)

b. Input(s): p, function (boolean)
t, relation (sequence)
Output: relation (sequence)

c. Description: Filters the relation t, using the predicate, p. Reconnects nodes that could be lost by the normal filtering discussed later in this section. Used as a part of the filtering function for data structures, is discussed next.

d. Example(s): Suppose the user wanted to eleminate the negative nodes of the below sequence:

```
?> s == (seq 3 4 -2 6 7 -1 2 -4]
?> ((rsec >= 0) xi s]
 (rel (3 4) (4 6) (6 7) (7 2)]
```

2. Filtering Data Structures

a. Syntax: (p PHI S)

b. Input(s): p, function (boolean)
S, relation (data structure)
Output: relation (data structure)

c. Description: Extends the 'xi' functional to work on RPL data structures. Note that the data part is not changed, only the form part is filtered.

d. Example(s): Consider the sequence used in the 'xi' example as a RPL data structure:

```
?> S == (list (list 3 4 -2 6 7 -1 2 -4)
              (seqrange 1 to 8)]
?> ((rsec >= 0) PHI S]
 (rel  (1 (rel (1 3) (2 4) (3 -2) (4 6)
               (5 7) (6 -1) (7 2) (8 -4)))
       (2 (rel (1 2) (4 5) (2 4) (5 7))) )
```

Note: Sequence order doesn't matter in the form part.

3. Filtering Relations

a. Syntax: (p filter t)

b. Input(s): boolean function; relation
Output: relation

c. Description: Eliminates undesireable nodes from t by applying the predicate to each element of t. If the predicate succeeds, the element is left in the relation, otherwise it is removed. This functional is the basis for the restriction operators discussed next in this section.

d. Example(s): Consider the same sequence, s, used in the example of the 'xi' operator. This will illustrate that this filtering method can eliminate valid nodes and leave nodes disconnected in the case of sequences:

```
?> val s
 (rel (3 4) (4 -2) (-2 6) (6 7)
      (7 -1) (-1 2) (2 -4))
?> p x == ((hd x) >= 0) and (tl x) >= 0)]
?> (p filter s]
 (rel (3 4) (6 7))
```

4. Restriction - Domain, Range, Both

a. Syntax: (p -> t)
(t <- p)
(t restr p)

b. Input(s): boolean function; relation
Output: relation

c. Description: Returns a relation which restricts the domain, range or both the domain and range, respectively. This is accomplished by filtering the table using the predicate p on the appropriate members of each element of the relation.

d. Example(s): Consider the same sequence s, used in previous examples:

```
(1)  ?> val s
       (rel (3 4) (4 -2) (-2 6) (6 7)
            (7 -1) (-1 2) (2 -4))
     ?> ((rsec >= 0) -> s]
       (rel (3 4) (4 -2) (6 7) (7 -1) (2 -4))

(2)  ?> (s <- (rsec >= 0)]
       (rel (3 4) (-2 6) (6 7) (-1 2))

(3) ?> (s restr (rsec >= 0)]
     (rel (3 4) (6 7))
```

5. Application

a. Syntax: (f @ x)

b. Input(s): function; anything
Output: anything

c. Description: Returns the result of applying f to the argument x.

d. Example(s):

```
?> ((op times) @ (list 2 3)]
 6
```

6. Application, Functional Record

a. Syntax: (t @hat x)

b. Input(s): relation (table of functions) anything
Output: relation

c. Description: Produces a relation which pairs each left member of the input relation to the result of applying the right member function to the argument x.

d. Example(s): Consider the following simple list of functions:

```
?> t == (list (op times) (op +) (op -) (op /)]
?> (t @hat (list 4 3]
 (rel (1 12) (2 7) (3 1) (4 1))
```

7. Application, Functional Structure

a. Syntax: (t ! x)

b. Input(s): relation
Output: relation

c. Description: The input table to this functional must have a domain and range which consists of functions only. The argument x must be valid for all functions contained within the table. Each element of t will be replaced by the result of applying both the left member and right member functions to the argument.

d. Example(s):

```
?>  t == (rel ((op times) : (op /))
              ((op +) : (op -))]
?> (t ! (list 4 3]
 (rel (12 1) (7 1))
```

8. Image of Sets

a. Syntax: (f img t)

b. Input(s): function; relation
Output: set

c. Description: Returns a set which is the result of applying f to every member of the set or relation t.

d. Example(s):

```
?> sqr == (times o DELTA]
?> (sqr img (set 1 2 3 4 5]
 (set 1 4 9 16 25)
```

9. Isomorphism, Image on Relations

a. Syntax: (f $ t)

b. Input(s): function; relation
Output: relation

c. Description: Returns a relation which has the same structure as the original, except that each element is composed of the result of applying f to both the left and right member of the element of t.

d. Example(s): Consider again the 'sqr' function:

```
?> (sqr ‡ (seqrange 1 to 6]
 (rel (1 4) (4 9) (9 16) (16 25) (25 36))
```

10. Relative Product, Intensional

a. Syntax: (t rp f)

b. Input(s): function
Output: relation

c. Description: Returns a relation which is the result of applying the function f to every right member of the input relation.

d. Example(s):

```
?> t == (listrange 1 to 5]
?> (t rp (rsec times 10]
 (rel (1 10) (2 20) (3 30) (4 40) (5 50))
```

11. Relative Product Inverse, Intensional

a. Syntax: (f rpi t)

b. Input(s): function
Output: relation

c. Description: Returns a relation which is the result of applying the function f to every left member of the input relation.

d. Example(s):

```
?> t == (listrange 1 to 5]
?> ((rsec times 10) rpi t]
 (rel (10 1) (20 2) (30 3) (40 4) (50 5))
```

12. Restriction of a Function

a. Syntax: (s restrict f)

b. Input(s): relation (set); function
Output: relation

c. Description: Transforms the function into a extensional relation (table) based upon the set of domain elements given as input. It pairs each element of s with the result of applying the function f to it.

d. Example(s): Suppose the user wanted a table of squares for the subrange 4 to 8:

```
?> s == (setrange 4 to 8]
?> sqr == (times o DELTA]
?> (s restrict sqr]
 (rel (4 16) (5 25) (6 36) (7 49) (8 64))
```

APPENDIX D - INDEX TO RPL OPERATORS

Name	Operator	Page
Addition	+	131
All	all	139
Application		
anything	@	157
functional record	@hat	158
functional structure	!	158
Array		
concatenation	cat	146
from sequence	sa	145
reduction	red	149
reverse	rev	146
Array to sequence	as	146
Bi-univalent	bun	142
Cardinality	size	135
Cartesian product	cart	135
Composition		
functions	o	149
repeat using superscription	sup n	127
Concatenation - array	cat	146
Conditional functional	if	126
Conjuntion	andsign, and	132
Cons left - sequence	cl	144
Cons right - sequence	cr	144
Construction	#	137
Converse		
relation	cnv	137
using superscription	sup -1	127
Curry		
extensional	cur	138
intensional	curry	150
Data definition		
list	list	123
list range	listrange	123
relation	rel	122
sequence	seq	122
sequence range	seqrange	123
set	set	122
set range	setrange	123
Data structures		
filtering	PHI	155
image	PI	154
operate on data part	delta	153

Data structures, cont.		
operate on form part	phi	154
value of a node	upsilon	153
Database		
index	index	147
join	join	148
select	select	147
Difference	-	131
Disjuntion	orsign, or	132
Division	divide or /	131
Domain	dom	140
Duplication	DELTA	129
Element selection	epsilon	133
Empty set or relation	empty	128
Equality	=	129
Extension of a relation	extend	150
Filtering		
data structures	PHI	155
relations	filter	156
sequences	xi	155
Final sequence	OMEGA	143
First member - sequence	alpha	143
Formalization functional	bar	128
Function definition		
conditional	if	126
direct	func	124
fix left argument	lsec	125
fix right argument	rsec	125
infix to prefix	op	125
iteration	iter	126
while loop	while	152
Greater	>	132
Greater or equal	>=	132
Head - elementary pair	hd	130
Identity	I	130
Image		
data structure	PI	154
of domain element	unimg	139
of range element	all	139
relations - isomorphism	$	159
sets	img	158
Improper subset	!subset	136
Inequality	!= or <>	129
Infix to prefix conversion	op	125
Intial members	init	141
Initial sequence	ALPHA	143
Intersection	cap	134
Isomorphism - relations	$	159
Iteration functional	iter	126
Last member - sequence	omega	143
Left member	Lm	142

Left section	lsec	125
Left univalent	lun	142
Less	<	132
Less or equal	<=	132
List definition	list	123
List range definition	listrange	123
Maximum - set	max	133
Member	Mm	142
Members	mem	141
Membership.	member	135
Minimize sequence	mu	144
Minimum - set	min	133
Multiplication	times	131
Negation	not	132
Negation - function	wig	151
Nonmembership	nomem	135
Ordered union	;	138
Pair Formation	:	130
Pair list	,	131
Paralleling	!!	151
Product	times	131
Proper subset	subset	136
Quotient	divide or /	131
Range	rng	140
Reduction		
array	red	149
Reflexive transitive closure	sup **	127
Relation definition	rel	122
Relational sort	rsort	133
Relative product		
intensional	rp	159
intensional inverse	rpi	160
primitive	!	139
Restriction		
domain	<-	157
of a function	restrict	160
range	->	157
range and domain	restr	157
Reverse array	rev	146
Right member	Rm	142
Right section	rsec	125
Right univalent	run	142
Selection	sel	136
Sequence		
all but first element	OMEGA	143
all but last element	ALPHA	143
cons left	cl	144
cons right	cr	144
convert to array	sa	145
convert to matrix	ssm	145
definition	seq	122

Sequence, cont.		
filtering	xi	155
first member	alpha	143
from array	as	146
last member	omega	143
minimize	mu	144
range definition	seqrange	123
Set definition	set	122
Set difference	\	134
Set range definition	setrange	123
Sort	sort	133
Subtraction	-	131
Sum	+	131
Superscription		
converse	sup -1	127
reflexive transitive closure	sup **	127
repeat composition	f sup n	127
transitive closure	sup +	127
Tail - elementary pair	tl	130
Terminal members	term	141
Transitive closure	sup +	127
Uncurry		
extensional	unc	138
intensional	uncurry	150
Union	cup	134
Unique element selection	theta	134
Unique set	uset	134
Unit image	unimg	139
Unit set	un	131
While loop	while	152

APPENDIX E - RPL INPUT FORM SUMMARY

TABLE 1. Primitive Extensional Operations

Name	Old Input Form	New Input Form	Publication Form
selection	t sel x	t sel x	$t \downarrow x$
relative product	t | u	t | u	$t \mid u$
construction	t , bar u	t # u	$t \# u$
pair formation	x : y	x : y	$x : y$
union	t cup u	t cup u	$t \cup u$
unit set	un x	un x	**un** x
currying	cur t	cur t	**cur** t
uncurrying	unc t	unc t	**unc** t
unique element selection	theta s	theta s	$\theta\ s$
element selection	(added)	epsilon t	$\epsilon\ t$
cardinality	size t	size t	size t
structure	str t	(deleted)	(deleted)
transitive closure	t sup +	t sup +	t^{+}
empty set	empty	empty	$\emptyset$

TABLE 2. Nonprimitive Extensional Operations: Group 1

Name	Old Input Form	New Input Form	Publication Form
pair list	(x, y)	(x , y)	(x, y)
left pair section	(x,)	(deleted)	(deleted)
right pair section	(,y)	(deleted)	(deleted)
duplication	DELTA x	DELTA x	$\Delta\ x$
membership	x member t	x member t	$x \in t$
nonmembership	x nomem t	x nomem t	$x \notin t$
improper subset	s !subset t	s !subset t	$s \subseteq t$
proper subset	s subset t	s subset t	$s \subset t$
equality	s = t	s = t	$s = t$
converse	inv t, t sup -1	cnv t, t sup -1	**cnv** t, t^{-1}
domain	dom t	dom t	**dom** t
range	rng t	rng t	**rng** t
members	mem t	mem t	**mem** t
left member	Lm (x,t)	x Lm t	x **Lm** t
right member	Rm (x,t)	x Rm t	x **Rm** t
member	Mm (x,t)	x Mm t	x **Mm** t
right univalent	run t	run t	**run** t
left univalent	lun t	lun t	**lun** t
bi-univalent	bun t	bun t	**bun** t
initial members	init t	init t	**init** t
terminal members	term t	term t	**term** t
reflexive transitive closure	t sup *	t sup **	$t^{\cdot}$
domain restriction	p -> t	p -> t	$p \rightarrow t$
range restriction	t <- p	t <- p	$t \leftarrow p$
restriction	t restr p	t restr p	$t \upharpoonright p$
sequence filtering	(added)	p xi t	$p\ \xi\ t$

TABLE 3. Nonprimitive Extensional Operations: Group 2

Name	Old Input Form	New Input Form	Publication Form
first member	alpha t	alpha t	$\alpha\ t$
last member	omega t	omega t	$\omega\ t$
initial sequence	ALPHA t	ALPHA t	A t
final sequence	OMEGA t	OMEGA t	$\Omega\ t$
ordered union	t ; u	t ; u	t ; u
cons left	x cl t	x cl t	x **cl** t
cons right	t cr x	t cr x	t **cr** x
minimum	min s	min s	**min** s
maximum	max s	max s	**max** s
intersection	s cap t	s cap t	$s \cap t$
set difference	s \ t	s \ t	$s \setminus t$
apply functional record	t @ hat x	t @ hat x	t @ x
apply functional structure	t ! x	t ! x	t ! x
minimize	mu t	mu t	$\mu\ t$
database index	index x d	x index d	x **index** d
database select	select x	x select d	x **select** d
database join	join x	x join dblist	x **join** *dblist*
array to sequence	as t	as t	**as** t
sequence to array	sa t	t sa i	t **sa** i
seq. to zero-origin array	sa0 t	(deleted)	(deleted)
relative product	rp f t	t rp f	$t \mid f$
relative product inverse	rpi f t	f rpi t	$f \mid t$
array concatenation	t cat u	t cat u	t **cat** u
relation sort	rsort s	rsort s	**rsort** s
sort	sort s	sort s	**sort** s
unit image	unimg t x	t unimg x	t **unimg** x
all	all t	all t	**all** t
sequence to matrix	ssm t	ssm t	**ssm** t

TABLE 4. Primitive Intensional Operations

Name	Old Input Form	New Input Form	Publication Form
application	f @ x	f @ x	f @ x
image	img f s	f img s	f **img** s
composition	f . g	f o g	$f \circ g$
infix to prefix	(added)	(op +), (op times), ...	$[+]$, $[\times]$, $\cdots$
left section	(x+), (x-), ...	(lsec x +), ...	$[x+]$, $[x-]$, $\cdots$
right section	(+ y), (-y), ...	(rsec + y), ...	$[+y]$, $[-y]$, $\cdots$
paralleling	f || g	f || g	$f \parallel g$
isomorphism	f $ t	f $ t	f $ t
formal application	f @ bar g	(deleted)	(deleted)
functional condition	(p -> f; g)	(if p -> f ; g)	$(p \rightarrow f; g)$
curry	curry f	curry f	**curry** f
uncurry	uncurry f	uncurry f	**uncurry** f
filtering	PHI p (d, r)	p PHI S	$p\ \Phi\ S$
iteration	iter [p -> f]	(iter p -> f)	**iter** $[p \rightarrow f]$
formalization	+ bar, times bar, ...	(+ bar), (times bar), ...	$\overline{+}$, $\overline{\times}$
identity	Id	I	**I**

TABLE 5. Nonprimitive Intensional Operations

Name	Old Input Form	New Input Form	Publication Form
while loop	while [p, f]	(f while p)	f **while** p
array reduction	f red i	f red x	$f \S x$
repeated composition	f sup n	f sup n	f^n
value of node	upsilon f	upsilon f	υf
operate on form	phi f	phi f	ϕf
operate on data	delta f	delta f	δf
image of structure	PI f	PI f	Πf
extension	extend (t, f)	t extend f	t **extend** f
restriction	restrict (s, f)	s restrict f	s **restrict** f
formal negation	wig p	wig p	$\sim p$

TABLE 6. Miscellaneous Operations

Name	Old Input Form	New Input Form	Publication Form
sum	x + y	x + y	$x + y$
difference	x - y	x - y	$x - y$
product	x times y	x times y	$x \times y$
quotient	x divide y	x divide y	$x \div y$
inequality	x != y	x != y	$x \neq y$
less	x < y	x < y	$x < y$
greater	x > y	x > y	$x > y$
less or equal	x <= y	x <= y	$x \leqslant y$
greater or equal	x >= y	x >= y	$x \geqslant y$
conjunction	x andsign y	x andsign y	$x \wedge y$
disjunction	x orsign y	x orsign y	$x \vee y$
negation	not x	not x	$\neg x$
cartesian product	s cart t	s cart t	$s \times t$

TABLE 7. Data Input Operations and Syntax

Name	Input Form	Publication Form
identifiers	a, b', total, etc.	a, b', total, *etc.*
strings	"abcd"	"abcd"
booleans	true, false	**true**, **false**
relation	(rel (x : y), ...)	$((x\ y), \cdots)$
set	(set x y ...)	$\{x, y, \cdots\}$
sequence	(seq x y ...)	$(x, y, \cdots)$
list	(list x y ...)	$< x, y, \cdots >$
subrange set	(setrange m to n)	$\{m, \ldots, n\}$
subrange sequence	(seqrange m to n)	$(m, \ldots, n)$
subrange list	(listrange m to n)	$< m, \ldots, n >$

TABLE 8. RPL Command Types

Name	Input Form	Publication Form
data definition	x = = y	$x \equiv y$
prefix function definition	f x = = y	$f\ x \equiv y$
infix function definition	x f y = = z	$x\ f\ y \equiv z$
write data to a file	file "name" = = x	**file** *"name"* $\equiv x$
read data from a file	x = = (file "name")	$x \equiv$ **file** *"name"*
output, form 1	display x	**display** x
output, form 2	dis x	**display** x
output, form 3	d x	**d** x
output, form 4	x	x
output value of definition	val x	**val** x
output function environment	env f	**env** f
output entire environment	env	**env**

APPENDIX F - RPL CODE AND DOCUMENTATION

```
(RPL
*****************************************************************
  calls:      INIT_SYS_NAMES, WRITE, MAPCAR, SET_USER_ENV,
              TERPRI, PRIN1, SPACES, CONS, READCMD, EXECUTE
  uses free:  BUILT_IN_PREFIX_OPS, INTOPS, SYSOPS, CMD,
              USERDEFS, SYSTEM_ENV, PREFIX_OPNAMES, OPNAMES,
              TEMPNAMES, ERRORCODE, E, FILTER_ON
  comments:   Shell for RPL Interpreter.
*****************************************************************
  (LAMBDA NIL
    (PROG NIL
          (INIT_SYS_NAMES)
          (SETQ FILTER_ON NIL)
          (WRITE (QUOTE (Loading RPL---)))
          (SETQ E SYSOPS)
          (SETQ ERRORCODE (QUOTE ERRORFREE))
          (SETQ TEMPNAMES OPNAMES)
          (SETQ OPNAMES NIL)
          (MAPCAR INTOPS (QUOTE EXECUTE))
          (SETQ OPNAMES TEMPNAMES)
          (SETQ PREFIX_OPNAMES
           BUILT_IN_PREFIX_OPS)
          (SETQ E (CONS (CONS (QUOTE SYSTEM)
                              (QUOTE SYSTEM)) E))
          (SETQ SYSTEM_ENV E)
          (SETQ USERDEFS NIL)
          (SET_USER_ENV)
          (TERPRI)
          (TERPRI)
          (WRITE (QUOTE (RPL INTERPRETER ON LINE!!)))
          (TERPRI)
          (TERPRI)
      LOOP(SETQ ERRORCODE (QUOTE ERRORFREE))
          (PRIN1 (QUOTE ?>))
          (SPACES 1)
          (SETQ CMD (READCMD))
          (TERPRI)
          (EXECUTE CMD)
          (GO LOOP])
```

```
(INIT_SYS_NAMES
************************************************************
  called by: RPL
  uses free: EMSG, SETOP, NUMOP, SPECIAL_CASES, SETS,
             INTOPS, BIFTAG_INFIX, SYSOPS, PREFIX_OPNAMES,
             BUILT_IN_PREFIX_OPS, OPNAMES, USERDEFS
  comments:  Initialization required to execute RPL.
************************************************************
  [LAMBDA NIL
    (SETQ USERDEFS NIL)
    (SETQ OPNAMES
      (QUOTE (SYSTEM done file display dis val env sup rel
              set seq list setrange seqrange listrange func
              empty true false filter hd tl lsec rsec op if
              iter or and <> # @ o $ red img curry uncurry
              PHI I while upsilon phi delta PI sel %! , :
              extend restrict wig cup member nomem !subset
              subset = -> <- restr ; cl cr cap \ @hat ! cat
              + - times divide / != < > <= >= andsign orsign
              cart un cur unc theta epsilon size DELTA cnv
              rev dom rng mem Lm Rm Mm run lun bun init term
              alpha omega ALPHA OMEGA min max uset mu index
              select join as sa rp rpi rsort sort unimg all
              ssm not PHIaux xi)))
    (SETQ BUILT_IN_PREFIX_OPS
      (QUOTE (lsec rsec op if iter hd tl un cur unc size
              theta epsilon DELTA cnv rev dom rng mem run
              lun bun init term alpha omega ALPHA OMEGA min
              max mu select join as sa rsort sort all ssm
              curry uncurry I while upsilon phi delta PI
              wig not uset)))
    (SETQ PREFIX_OPNAMES NIL)
    [SETQ SYSOPS (QUOTE ((unimg closure select_all)
                         (hd closure Hd)
                         (tl closure Tl)
                         (filter closure filter)
                         (run closure run)
                         (# closure construction)
                         (size closure cardinality)
                         (rpi closure rel_prod_inv)
                         (rp closure rel_prod)
                         (img closure img)
                         (empty Eset)
                         (true true)
                         (false false)
                         (+ closure +)
                         (- closure -)
                         (times closure *)
                         (divide closure /)
                         (/ closure /)
                         (< closure <)
```

```
                        (> closure >)
                        (<= closure <=)
                        (>= closure >=)
                        (not closure not)
                        (or closure or)
                        (and closure and)
                        (orsign closure or)
                        (andsign closure and)
                        (epsilon closure elementselect)
                        (theta closure unitset_select)
                        (un closure unitset)
                        (cup closure union)
                        (cap closure intersection)
                        (\ closure setdiff)
                        (cart closure cart)
                        (subset closure subset)
                        (!subset closure !subset)
                        (= closure =)
                        (!= closure <>)
                        (<> closure <>)
                        (member closure member)
                        (nomem closure nomem)
                        (%! closure %!)
                        (file closure file)
                        (sel closure sel)
                        (: closure :)
                        (dom closure dom)
                        (rng closure rng)
                        (cnv closure converse)
                        (sup closure superscript)
                        (rev closure reverse_array)
                        (** closure star)
                        ($ closure isomorphism)
                        (as closure array_to_seq)
                        (sa closure seq_to_array)
                        (min closure min)
                        (max closure max)
                        (cat closure concatenation)
                        (cur closure curry_ext)
                        (unc closure uncurry_ext)
                        (rsort closure rsort)
                        (sort closure sort)
                        (red closure reduction)
                        (uset closure unique_set]
(SETQ BIFTAG_INFIX
  (QUOTE (+ - * / < > <= >= or and union intersection
          setdiff cart subset !subset = <> member nomem
          construction %! sel : img rel_prod rel_prod_inv
          filter superscript isomorphism concatenation
          seq_to_array select_all reduction)))
```

```
[SETQ INTOPS (QUOTE ([o ==(func (f g) (func x (f (g x]
                    (lun t ==((run o cnv) t))
                    (bun t ==((run t) and (lun t)))
                    (x Rm t ==(x member (rng t)))
                    (x Lm t ==(x member (dom t)))
                    (mem t ==((dom t) cup (rng t)))
                    (term t ==((rng t) \ (dom t)))
                    (x Mm t ==(x member (mem t)))
                    (init t ==((dom t) \ (rng t)))
                    (t <- p ==((p o tl) filter t))
                    (p -> t ==((p o hd) filter t))
                    (t restr p ==((p -> t) <- p))
                    (t ; u ==(t cup ((rsec member
                     ((dom u) \ (dom t))) -> u)))
                    (alpha t ==((theta o init) t))
                    (omega t ==((theta o term) t))
                    [ALPHA s ==(s <-(rsec nomem (term s]
                    (OMEGA t ==((rsec nomem (init t))
                              -> t))
                    (x cl t ==((rel (x :(alpha t)))
                        cup t))
                    [t cr x ==(t cup (rel ((omega t): x]
                    (f @ x ==(f x))
                    (x , y ==(list x y))
                    [%!%! ==(func (f g) (func (x y)
                                  (list (f x) (g y]
                    (I x == x)
                    (wig p ==(not o p))
                    (DELTA x ==(list x x))
                    (phi ==(lsec I %!%!))
                    (delta ==(rsec %!%! I))
                    (f while p ==(if p -> (iter p -> f)
                                     ; I))
                    (PI f ==(delta (rsec rp f)))
                    (upsilon f ==
                       (sel o (I %!%! f)))
                    (t extend f ==
                       (if (rsec member (dom t))
                           -> (lsec t sel) ; f))
                    (s restrict f ==(((op :)
                       o ((I %!%! f) o DELTA)) img s))
                    (x index t ==(((rsec sel x)
                         (: bar) I) img t))
                    (t @hat x ==
                       ((hd (: bar) ((rsec @ x) o tl))
                        img t))
                    (t ! x ==((rsec @ x) ‡ t))
                    [mu t ==(t \ (t %! (t sup +]
                    (p xi r ==(mu ((r sup +) restr p)))
                    [t PHIaux s ==((s sel 1) ,
                        ((rsec Lm t) xi (s sel 2]
```

```
                        (p PHI s ==(((s sel 1) <- p)
                            PHIaux s))
                        (ssm t ==((unc o (rsec sa 1))
                              ((rsec sa 1) # t)))
                        (y all t ==((cnv t) unimg y))
                        [x select d ==
                           (rng o (rsec -> (x index d]
                        (x join dp ==
                           ([(lsec (cup o (hd (, bar) tl)) img)
                             o (((rsec sel 1)
                                 (%! bar) (rsec sel 2))
                               o ((cnv %!%! I)
                                 o ((lsec x index)
                                    %!%! (lsec x index]
                            dp))
                        [curry f ==(func x (func y (f (x , y]
                        (uncurry f ==(func (x y) ((f x) y]
(SETQ SETS (QUOTE (rel set setrange seq seqrange list
                   listrange)))
(SETQ SPECIAL_CASES
  (QUOTE (Eset Erel rel set setrange seq seqrange list
          listrange op lsec rsec func if when iter
          repeat reduce)))
(SETQ NUMOP (QUOTE (+ - * / < > <= >=)))
(SETQ SETOP (QUOTE (cart union intersection setdiff
                    subset !subset)))
(SETQ EMSG (QUOTE ((BAD_CMD bad command)
                   (UBI unbound individual)
                   (PARAM number of parameters in error)
                   (BAD_RANGE bad range variables)
                   (BAD_SEQ bad sequence)
                   (EXP_SET set, relation, sequence
                            or list expected)
                   (EXP_SEQ sequence expected)
                   (EXP_NUM numeric arguments expected)
                   (EXP_REL relation expected)
                   (UBTE unbound table element)
                   (EXP_FUNC function expected)
                   (UDF undefined function)
                   (BAD_SYNTAX syntax error)
                   (EXP_BOOL boolean predicate expected)
                   (BAD_ARGS invalid arguments)
                   (EXP_UNITSET unit set expected)
                   (EXP_INFIX infix operator expected)
                   (EXP_PAIR elementary pair expected)
                   (EXP_NSET numeric set expected)
                   (EXP_ARRAY array expected)
                   (ZERO_DIV zero divisor)
                   (EXP_NEMPTY non-empty set expected)
                   (BIF built in function or RPL keyword]))
```

```
(SET_USER_ENV
**********************************************************************
  calls:      MEMBER, WRITE, TERPRI, READ_USER_DEFS, READTERM
  called by: RPL, EXIT
  binds:      RESP, FILENAME
**********************************************************************
  [LAMBDA NIL
    (PROG (RESP FILENAME)
          (WRITE (QUOTE (DO YOU WANT TO RESUME A PREVIOUS
                         RPL SESSION? <y/n>?)))
          (SETQ RESP (READTERM))
          (COND
           ((MEMBER RESP (QUOTE (y Y)))
             (WRITE (QUOTE (INPUT FILENAME)))
             (TERPRI)
             (SETQ FILENAME (READTERM))
             (READ_USER_DEFS FILENAME]

(EXECUTE
**********************************************************************
  args:       CMD
  calls:      MEMBER, POSIT, LENGTH, DEF_BINDING, FILE_WRITE,
              EV, DISPLAY, EXIT, LIST, CONS, ERROR_HANDLER
  called by: READ_USER_DEFS, RPL
  binds:      X
  uses free: E
  comments:  Command level parser.
**********************************************************************
  [LAMBDA (CMD)
    (PROG (X)
          (SETQ X (POSIT CMD (QUOTE ==)))
          (RETURN (COND
                    ((AND (EQ X 2)
                          (EQ (LENGTH CMD) 3))
                      (DEF_BINDING CMD))
                    [(AND (EQ X 3)
                          (EQ (LENGTH CMD) 4))
                      (COND
                        ((EQ (CAR CMD) (QUOTE file))
                          (FILE_WRITE (EV (CADR CMD) E)
                            (EV (CADDDR CMD) E)))
                        (T (DEF_BINDING CMD]
                    ((AND (EQ X 4)
                          (EQ (LENGTH CMD) 5))
                      (DEF_BINDING CMD))
                    [(EQ X 0)
                      (COND
                        ((AND (MEMBER (CAR CMD)
                                  (QUOTE (display dis d env val)))
                              (EQ (LENGTH CMD) 2))
                          (DISPLAY CMD))
```

```
                        ((EQ (CAR CMD) (QUOTE done)) (EXIT))
                        [(EQ (LENGTH CMD) 1)
                          (COND
                            ((EQ (CAR CMD) (QUOTE env))
                              (DISPLAY (LIST (QUOTE env) NIL)))
                            (T (DISPLAY (CONS (QUOTE d) CMD]
                        (T (ERROR_HANDLER (QUOTE BAD_CMD) CMD]
                    (T (ERROR_HANDLER (QUOTE BAD_CMD) CMD]))

(DEF_BINDING
*****************************************************************
  args:       DEXP
  calls:      MEMB, ERROR_HANDLER, LDIFFERENCE, SPACES,
              WRITE, LENGTH; LOOKUP, CONS, LIST, EV, LAST,
              SASSOC, TERPRI, RPLACD, READTERM
  called by:  EXECUTE
  binds:      NAME, NEWNAME, EXP, RESP
  uses free:  ERRORCODE, OPNAMES, USERDEFS, PREFIX_OPNAMES, E
  comments:   Makes all bindings to the environment; includes
              mechanism to implement simple recursion.
*****************************************************************
  [LAMBDA (DEXP)
    (PROG (NAME EXP NEWNAME RESP)
          [COND
           ((EQ (LENGTH DEXP) 5) (SETQ NAME (CADR DEXP)))
           (T (SETQ NAME (CAR DEXP]
          [COND
           ((MEMB NAME OPNAMES)
             (WRITE (QUOTE (SYSTEM DEFINED FUNCTION OR
                            KEYWORD, OVERWRITE? <y/n>)))
             (SETQ RESP (READTERM)) (TERPRI)
             (COND
              ([NOT (MEMB RESP (QUOTE (Y y]
                (WRITE (QUOTE (ABORT AT USER'S REQUEST)))
                (TERPRI) (TERPRI) (GO EXIT]
           [COND
            ((EQ (LOOKUP NAME E) NIL)
              (SETQ NEWNAME NIL)
              (SETQ E (CONS (CONS NAME NIL) E)))
            (T (SETQ NEWNAME T]
          [COND
           ((EQ (LENGTH DEXP) 4)
             (SETQ EXP (LIST (QUOTE closure)
                             (CADR DEXP)
                             (CADDDR DEXP) E)))
           ((EQ (LENGTH DEXP) 5)
             (SETQ EXP (LIST (QUOTE closure)
                             (LIST (CAR DEXP)
                                   (CADDR DEXP))
                             (CADDDR (CDR DEXP)) E)))
           (T (SETQ EXP (EV (CAR (LAST DEXP)) E]
```

```
          BIND_NAME
          (RETURN (COND
                    [(EQ ERRORCODE (QUOTE ERRORFREE))
                      (COND
                        ((EQ NEWNAME T)
                          [COND
                            ((AND (MEMB NAME PREFIX_OPNAMES)
                                  (EQ (LENGTH DEXP) 5))
                              (LDIFFERENCE PREFIX_OPNAMES
                                           (LIST NAME)))
                            ((AND (NOT (MEMB NAME PREFIX_OPNAMES))
                                  (NOT (EQ (LENGTH DEXP) 5)))
                              (SETQ PREFIX_OPNAMES
                                (CONS NAME PREFIX_OPNAMES]
                          (COND
                           ((NOT (MEMB NAME OPNAMES))
                             (RPLACD (SASSOC NAME E) EXP)
                          (RPLACD (SASSOC NAME USERDEFS) DEXP)
                          (SPACES 1)
                          (WRITE (LIST NAME (QUOTE Redefined)))
                          (TERPRI) (TERPRI))
                         (T (SETQ USERDEFS
                              (CONS (CONS NAME DEXP) USERDEFS))
                            [COND
                              ((AND (LISTP EXP) (EQ (CAR EXP)
                                (QUOTE closure)))
                                (COND
                                  ((OR (EQ (LENGTH DEXP) 4)
                                       (EQ (LENGTH DEXP) 3))
                                    (SETQ PREFIX_OPNAMES
                                      (CONS NAME PREFIX_OPNAMES]
                             (RPLACD (SASSOC NAME E) EXP]
                    (T (WRITE (QUOTE (BINDING CANNOT BE MADE)))
                       (TERPRI) (TERPRI)
                       (COND
                         ((NOT (EQ NEWNAME T)) (SETQ E (CDR E]
     EXIT])


(DISPLAY
*****************************************************************
  args:       CMD
  calls:      MEMBER, LITATOM, PRINT, SHOW_ATOM, TERPRI,
              DISPLAY_ENV, ERROR_HANDLER, LOOKUP, EV, TYPE,
              LENGTH
  called by:  EXECUTE
  binds:      KEY, EXP, EVEXP
  uses free:  ERRORCODE, E, USERDEFS
  comments:   Performs all output to the screen to include
              the "val" and "env" commands.
*****************************************************************
```

```
[LAMBDA (CMD)
  (PROG (KEY EXP EVEXP)
        (SETQ KEY (CAR CMD))
        (SETQ EXP (CADR CMD))
        [COND
         [(MEMBER KEY (QUOTE (d dis display)))
           (COND
             [(LITATOM EXP)
               (SETQ EVEXP (LOOKUP EXP USERDEFS))
               (COND
                 ((NULL EVEXP)
                   (PRINT (QUOTE Undefined)))
                 (T (PRINT EVEXP]
             (T (SETQ EVEXP (EV EXP E))
                (COND
                  ((NULL EVEXP)
                    (PRINT (QUOTE Undefined)))
                  (T (SHOW_ATOM EVEXP)
                     (TERPRI]
         (T [COND
              ((NOT (NULL EXP))
                (SETQ EVEXP (EV EXP E]
            (COND
              ((EQ ERRORCODE (QUOTE ERRORFREE))
                (COND
                  [(AND (EQ KEY (QUOTE val))
                        (LITATOM (CADR CMD)))
                    (COND
                      ((NULL EVEXP)
                        (PRINT (QUOTE Undefined)))
                      (T (SHOW_ATOM EVEXP)
                         (TERPRI]
                  ((AND (EQ KEY (QUOTE env))
                        (NULL EXP))
                    (DISPLAY_ENV EXP))
                  ((AND (EQ KEY (QUOTE env))
                        (LITATOM (CADR CMD))
                        (EQ (TYPE EVEXP)
                            (QUOTE closure))
                        (EQ (LENGTH EVEXP) 4))
                    (DISPLAY_ENV EXP))
                  ((EQ KEY (QUOTE env))
                    (ERROR_HANDLER
                      (QUOTE EXP_FUNC) CMD))
                  (T (ERROR_HANDLER
                       (QUOTE BAD_SYNTAX) CMD]
        (TERPRI])
```

```
(ERROR_HANDLER
**************************************************************
  args:       CODE, EXP
  calls:      WRITE, TERPRI, PRINT_LIST, LOOKUP
  called by:  DEF_BINDING, DISPLAY, EVRANGE, EVSEQ, RPAPPLY,
              ARRAY_REDUCTION, MIN_SET, RPL_REPEAT, EXECUTE,
              EV, EV_SPECIAL_CASES, INFIXOP, PREFIXOP,
              BIF_APPLY, ARRAY_CONCATENATION, HEAD, MAX_SET,
              MEM, SEL, SUPERSCRIPT, TAIL, BINARY_LIST,
              COERCE_TO_REL
  uses free:  EMSG, FILTER_ON, ERRORCODE
  comments:   Based on the CODE given, displays the appro-
              piate error message and the probable cause of
              error, EXP.
**************************************************************
  [LAMBDA (CODE EXP)
    (PROG NIL
          (COND
           ((EQ FILTER_ON T)
             (GO EXIT)))
          (WRITE (QUOTE (*** ERROR ***)))
          (WRITE (LOOKUP CODE EMSG))
          (TERPRI)
          (WRITE (QUOTE (Cause of error ==>)))
          (PRINT_LIST EXP)
          (TERPRI)
      EXIT(SETQ ERRORCODE CODE) NIL])

(EXIT
**************************************************************
  calls:      MEMBER, WRITE, TERPRI, WRITE_USER_DEFS,
              READTERM, SET_USER_ENV
  called by:  EXECUTE
  uses free:  BUILT_IN_PREFIX_OPS, SYSTEM_ENV, USERDEFS,
              PREFIX_OPNAMES, E, FILENAME, RESP
  comments:   Used to exit the RPL environment or begin
              another session.
**************************************************************
  [LAMBDA NIL
    (WRITE (QUOTE (DO YOU WANT TO SAVE ENVIRONMENT FOR
                   FUTURE USE? <y/n?>)))
    (SETQ RESP (READTERM))
    (COND
      ((MEMBER RESP (QUOTE (y Y)))
       (WRITE (QUOTE (INPUT FILENAME)))
       (TERPRI)
       (SETQ FILENAME (READTERM))
       (WRITE_USER_DEFS FILENAME)))
    (TERPRI)
```

```
    (WRITE (QUOTE (EXIT TO LISP - PRESS ^D)))
    (TERPRI)
    (WRITE (QUOTE (EXIT TO UNIX - PRESS ^C)))
    (TERPRI)
    (WRITE (QUOTE (CONTINUE RPL - PRESS <RETURN>)))
    (TERPRI)
    (READTERM)
    (TERPRI)
    (WRITE (QUOTE (DO YOU WANT TO CLEAR CURRENT
                   ENVIRONMENT? <y/n?>)))
    (SETQ RESP (READTERM))
    (TERPRI)
    (COND
      ((MEMBER RESP (QUOTE (y Y)))
       (SETQ E SYSTEM_ENV)
       (SETQ USERDEFS NIL)
       (SETQ PREFIX_OPNAMES BUILT_IN_PREFIX_OPS)))
    (SET_USER_ENV])


(EV
**********************************************************************
  args:       EXP, E
  calls:      NUMBERP, STRINGP, ATOM, MEMBER, LOOKUP,
              ERROR_HANDLER, EV_SPECIAL_CASES, LENGTH,
              PREFIXOP, INFIXOP
  called by:  EXECUTE, DEF_BINDING, DISPLAY, MAPEV,
              EV_SPECIAL_CASES, EVSEQ, INFIXOP, PREFIXOP,
              RPAPPLY, ARRAY_REDUCTION, RPL_REPEAT,
              MAKE_UNIQUE
  binds:      X, TAG
  uses free:  SPECIAL_CASES
  comments:   Given an expression, EXP, and its environment,
              E, this function directs its evaluation.
**********************************************************************
  [LAMBDA (EXP E)
    (PROG (X TAG)
          (RETURN (COND
                    ((NUMBERP EXP) EXP)
                    ((STRINGP EXP) EXP)
                    ((ATOM EXP)
                      (SETQ X (LOOKUP EXP E))
                      (COND
                        ((EQ X NIL)
                          (ERROR_HANDLER
                            (QUOTE UBI) EXP))
                        (T X)))
                    (T (SETQ TAG (CAR EXP))
                       (COND
                         ((MEMBER TAG SPECIAL_CASES)
                           (EV_SPECIAL_CASES EXP E))
```

```
                        ((EQ (LENGTH EXP) 2)
                          (PREFIXOP EXP E))
                        [(EQ (LENGTH EXP) 3)
                          (COND
                            ((LISTP (CADR EXP))
                              (EV_SPECIAL_CASES EXP E))
                            (T (INFIXOP EXP E]
                        (T (ERROR_HANDLER
                             (QUOTE PARAM) EXP])

(EV_SPECIAL_CASES
**********************************************************************
  args:        EXP, E
  calls:       MEMBER, ALL_PAIRS, ATOM, CONS, MAKE_UNIQUE,
               ERROR_HANDLER, LENGTH, EV, EVRANGE, EVSEQ,
               LIST, TYPE, RPL_REPEAT, ARRAY_REDUCTION
  called by:   EV
  binds:       TAG, LOW, HIGH, F
  uses free:   PREFIX_OPNAMES, ERRORCODE, SETS
  comments:    Handles all operators with special syntax.
**********************************************************************
  [LAMBDA (EXP E)
    (PROG (TAG LOW HIGH F)
          (SETQ TAG (CAR EXP))
          (RETURN (COND
                    [(MEMBER TAG SETS)
                      (COND
                        [(EQ TAG (QUOTE set))
                          (SETQ EXP (CONS (QUOTE Eset)
                                          (MAKE_UNIQUE
                                            (CDR EXP)
                                            NIL E]
                        ((EQ TAG (QUOTE rel))
                          (SETQ EXP (CONS (QUOTE Erel)
                                          (MAKE_UNIQUE
                                            (CDR EXP)
                                            NIL E)))
                          (COND
                            ((NOT (ALL_PAIRS (CDR EXP)))
                              (ERROR_HANDLER
                                (QUOTE EXP_REL) EXP))
                            (T EXP)))
                        [(EQ TAG (QUOTE setrange))
                          (COND
                            [(AND (EQ (LENGTH EXP) 4)
                                  (EQ (CADDR EXP)
                                      (QUOTE to)))
                              (SETQ LOW (EV (CADR EXP) E))
                              (SETQ HIGH (EV (CADDDR EXP) E))
                              (COND
```

```
                [(EQ ERRORCODE
                     (QUOTE ERRORFREE))
                  (SETQ EXP
                    (CONS (QUOTE Eset)
                          (EVRANGE LOW HIGH]
                (T (QUOTE impossible]
            (T (ERROR_HANDLER
                 (QUOTE BAD_RANGE) EXP]
        [(MEMBER TAG (QUOTE (seq seqrange)))
          (SETQ EXP (CONS (QUOTE Erel)
                          (EVSEQ EXP E]
        [(MEMBER TAG (QUOTE (list listrange)))
          (SETQ EXP (CONS (QUOTE Erel)
                          (EVSEQ EXP E]
        (T (QUOTE impossible]
((MEMBER TAG (QUOTE (Eset Erel))) EXP)
((EQ TAG (QUOTE func))
  (LIST (QUOTE closure)
        (CADR EXP)
        (CADDR EXP) E))
[(AND (EQ TAG (QUOTE op))
      (EQ (LENGTH EXP) 2))
  (SETQ F (EV (CADR EXP) E))
  (COND
    ([OR (NOT (EQ (TYPE F)
                  (QUOTE closure)))
         (AND (EQ (LENGTH F) 2)
              (MEMBER (CADR EXP)
                      PREFIX_OPNAMES))
         (AND (EQ (LENGTH F) 4)
              (ATOM (CADR F]
      (ERROR_HANDLER
        (QUOTE EXP_INFIX)
        (CADR EXP)))
    (T (LIST (QUOTE closure)
             (QUOTE ?x)
             (LIST (LIST (QUOTE ?x)
                         (QUOTE sel) 1)
                   (CADR EXP)
                   (LIST (QUOTE ?x)
                         (QUOTE sel) 2))
             E]
((AND (EQ TAG (QUOTE lsec))
      (EQ (LENGTH EXP) 3))
  (LIST (QUOTE closure)
        (QUOTE ?x)
        (LIST (CADR EXP)
              (CADDR EXP)
              (QUOTE ?x)) E))
((AND (EQ TAG (QUOTE rsec))
      (EQ (LENGTH EXP) 3))
```

```
    (LIST (QUOTE closure)
          (QUOTE ?x)
          (LIST (QUOTE ?x)
                (CADR EXP)
                (CADDR EXP)) E))
((AND (EQ TAG (QUOTE if))
      (EQ (LENGTH EXP) 6)
      (EQ (CADDR EXP) (QUOTE ->))
      (EQ (CADDDR (CDR EXP)) (QUOTE ;)))
    (LIST (QUOTE closure)
          (QUOTE ?x)
          (LIST (QUOTE when)
                (LIST (CADR EXP)
                      (QUOTE ?x))
                (QUOTE do)
                (LIST (CADDDR EXP)
                      (QUOTE ?x))
                (QUOTE elsedo)
                (LIST (CADDDR (CDDR EXP))
                      (QUOTE ?x))) E))
[(EQ TAG (QUOTE when))
  (COND
    ((EQ (EV (CADR EXP) E)
         (QUOTE true))
      (EV (CADDDR EXP) E))
    ((EQ (EV (CADR EXP) E)
         (QUOTE false))
      (EV (CADDDR (CDDR EXP)) E))
    (T (ERROR_HANDLER (QUOTE EXP_BOOL)
         (LIST (CADR EXP) (QUOTE in) EXP]
((AND (EQ TAG (QUOTE iter))
      (EQ (LENGTH EXP) 4)
      (EQ (CADDR EXP) (QUOTE ->)))
  (LIST (QUOTE closure)
        (QUOTE ?x)
        (LIST (QUOTE repeat)
              (CADDDR EXP)
              (QUOTE until_not)
              (CADR EXP) E))
((EQ TAG (QUOTE repeat))
  (RPL_REPEAT EXP E))
((AND (LISTP (CADR EXP))
      (EQ (LENGTH (CADR EXP)) 2)
      (EQ (CADADR EXP) (QUOTE bar)))
  (LIST (QUOTE closure)
        (QUOTE ?x)
        (LIST (LIST (CAR EXP)
                    (QUOTE ?x))
              (CAADR EXP)
              (LIST (CADDR EXP)
                    (QUOTE ?x))) E))
```

```
                    ((EQ TAG (QUOTE reduce))
                      (ARRAY_REDUCTION EXP E))
                    (T (ERROR_HANDLER
                         (QUOTE BAD_SYNTAX) EXP])

(MAPEV
**********************************************************************
  args:       L, E
  calls:      MAPCAR, EV
  called by:  EVSEQ
  binds:      X
  comments:   Given the list, L, and its environment, E, it
              returns a list of evaluated elements.
**********************************************************************
  [LAMBDA (L E)
    (MAPCAR L (QUOTE (LAMBDA (X) (EV X E])

(EVRANGE
**********************************************************************
  args:       LOW, HIGH
  calls:      NUMBERP, LEQ, ERROR_HANDLER, LIST, CONS,
              DIFFERENCE
  called by:  EV_SPECIAL_CASES, EVSEQ
  binds:      L
  comments:   Enumerates the range from LOW to HIGH and
              returns the list of numbers.
**********************************************************************
  [LAMBDA (LOW HIGH)
    (PROG (L)
          (SETQ L NIL)
          (COND
           ((AND (NUMBERP LOW)
                 (NUMBERP HIGH)
                 (LEQ LOW HIGH))
             (GO MAKE_LIST))
           (T (ERROR_HANDLER (QUOTE BAD_RANGE)
                (LIST LOW HIGH))
              (GO EXIT)))
         MAKE_LIST
          (COND
           ((EQ LOW HIGH) (SETQ L (CONS LOW L)))
           (T (SETQ L (CONS HIGH L))
              (SETQ HIGH (DIFFERENCE HIGH 1))
              (GO MAKE_LIST)))
      EXIT(RETURN L])
```

```
(EVSEQ
**************************************************************
  args:       SQ, E
  calls:      MEMBER, GREATERP, ERROR_HANDLER, LENGTH, MAPEV,
              EV, EVRANGE, SEQ_TO_REL, LIST_TO_REL
  called by:  EV_SPECIAL_CASES
  binds:      TAG, S, LOW, HIGH
  uses free:  ERRORCODE
  comments:   Takes the tagged sequence or sequence range,
              SQ, its environment, E, and returns a tagged
              evaluated relation.
**************************************************************
  [LAMBDA (SQ E)
    (PROG (TAG S HIGH LOW)
          (SETQ TAG (CAR SQ))
          (SETQ S (CDR SQ))
          [COND
           ((AND (MEMBER TAG (QUOTE (seq list)))
                 (GREATERP (LENGTH S) 1))
             (SETQ S (MAPEV S E))
             (GO COERCE))
           (T (COND
                ((AND (EQ (LENGTH S) 3)
                      (EQ (CADR S) (QUOTE to)))
                  (SETQ LOW (EV (CAR S) E))
                  (SETQ HIGH (EV (CADDR S) E))
                  (COND
                    ((EQ ERRORCODE (QUOTE ERRORFREE))
                      (SETQ S (EVRANGE LOW HIGH))
                      (GO COERCE))
                    (T NIL)))
                (T (ERROR_HANDLER
                     (QUOTE BAD_SEQ) S]
      COERCE
          (RETURN (COND
                    [(EQ ERRORCODE (QUOTE ERRORFREE))
                      (COND
                        ((MEMBER TAG (QUOTE (seq seqrange)))
                          (SEQ_TO_REL S))
                        (T (LIST_TO_REL S]
                    (T NIL])
```

```
(INFIXOP
**************************************************************
  args:       IEXP, ENV-I
  calls:      EV, TYPE, LIST, CONS, RPAPPLY, ERROR_HANDLER
  called by:  EV
  binds:      L, OP, R, A
  uses free:  ERRORCODE
  comments:   Performs pre-processing for evaluation.  The
              arguments, L and R, and operator, OP, are
              extracted from IEXP and evaluated in ENV-I, the
              environment.  The argument list is created and
              is sent with the operator to be evaluated.
**************************************************************
  [LAMBDA (IEXP ENV-I)
    (PROG (L OP R A)
          (SETQ L (EV (CAR IEXP) ENV-I))
          (SETQ OP (EV (CADR IEXP) ENV-I))
          (SETQ R (EV (CADDR IEXP) ENV-I))
          (RETURN (COND
                    ((EQ ERRORCODE (QUOTE ERRORFREE))
                      (COND
                        ((EQ (TYPE OP) (QUOTE closure))
                          (SETQ A (LIST (QUOTE Erel)
                                        (CONS 1 L)
                                        (CONS 2 R)))
                          (RPAPPLY OP A))
                        (T (ERROR_HANDLER (QUOTE EXP_FUNC)
                                          (CADR IEXP])

(PREFIXOP
**************************************************************
  args:       PEXP, ENV-P
  calls:      EV, TYPE, RPAPPLY, ERROR_HANDLER
  called by:  EV
  binds:      OP, ARG
  uses free:  ERRORCODE
  comments:   Same as INFIXOP, except for prefix operators.
**************************************************************
  [LAMBDA (PEXP ENV-P)
    (PROG (OP ARG)
          (SETQ OP (EV (CAR PEXP) ENV-P))
          (SETQ ARG (EV (CADR PEXP) ENV-P))
          (RETURN (COND
                    ((EQ ERRORCODE (QUOTE ERRORFREE))
                      (COND
                        ((EQ (TYPE OP) (QUOTE closure))
                          (RPAPPLY OP ARG))
                        (T (ERROR_HANDLER
                             (QUOTE EXP_FUNC) (CAR PEXP])
```

```
(RPAPPLY
**************************************************************
  args:       F, A  (Evaluated form)
  calls:      ATOM, ERROR_HANDLER, LENGTH, BIF_APPLY, CONS,
              DIFFERENCE, BINDARGS, APPEND, LIST, EV
  called by:  INFIXOP, PREFIXOP, ARRAY_REDUCTION, FILTER,
              MAPIMG, MAPRP, MAPRP_INV, MAP_ISOMORPHISM,
              RPL_REPEAT
  binds:      FORMALS, EE, LE
  uses free:  ERRORCODE
  comments:   Determines if F is a LISP defined function or
              intensionally defined function.  Evaluates the
              latter with the argument, A, and sends the
              former with argument to BIF_APPLY.
**************************************************************
  [LAMBDA (F A)
    (PROG (FORMALS LE EE)
          (RETURN (COND
                    ((EQ (LENGTH F) 2)
                      (BIF_APPLY F A))
                    (T (SETQ FORMALS (CADR F))
                       [COND
                         [(ATOM FORMALS)
                           (SETQ EE (CONS (CONS FORMALS A)
                                          (CADDDR F]
                         (T (COND
                              [(EQ (DIFFERENCE (LENGTH A) 1)
                                   (LENGTH FORMALS))
                                (SETQ LE (BINDARGS FORMALS A))
                                (SETQ EE (APPEND LE (CADDDR F]
                              (T (ERROR_HANDLER (QUOTE PARAM)
                                   (LIST (QUOTE (number of
                                                 parameters in error]
                       (COND
                         ((EQ ERRORCODE (QUOTE ERRORFREE))
                           (EV (CADDR F) EE])


(BINDARGS
**************************************************************
  args:       F, A
  calls:      MAP2CAR
  called by:  RPAPPLY
**************************************************************
  [LAMBDA (F A)
    (MAP2CAR F (CDR A)
             (QUOTE (LAMBDA (X Y) (CONS X (CDR Y])
```

```
(BIF_APPLY
**************************************************************
  args:        F, ARG
  calls:       MEMB, NUMBERP, ZEROP, ATOM, NUMERIC_SET,
               COERCE_TO_REL, TYPE, LENGTH, ERROR_HANDLER,
               SEL, LIST, PLUS, DIFFERENCE, TIMES, QUOTIENT,
               TF, GREATERP, LEQ, GEQ, CONS, INTERSECTION,
               UNION, LDIFFERENCE, CART_PROD, DO_SUBSET,
               REQUAL, RNOT, MEM, RELATIVE_PRODUCT,
               CONSTRUCTION, ARRAY_CONCATENATION,
               SEQ_TO_ARRAY, SELECT_ALL, MAPRP, FORM_PAIR,
               MAPIMG, MAPRP_INV, MAP_ISOMORPHISM, FILTER,
               SUPERSCRIPT, FILE_READ, CONVERSE, DOMAIN,
               RANGE, MAKE_UNIQUE, REVERSE_ARRAY,
               ARRAY_TO_SEQ, CURRY_EXT, UNCURRY_EXT, HEAD,
               TAIL, MIN_SET, MAX_SET, SEQ_TO_REL, SORT,
               LIST_TO_REL, LESSP
  called by:   RPAPPLY
  binds:       OP, L, R
  uses free:   ENV-P, PEXP, ERRORCODE, SETOP, IEXP, NUMOP,
               ENV-I, BIFTAG_INFIX
  comments:    Evaluates all built-in LISP defined operators.
**************************************************************
  [LAMBDA (F ARG)
    (PROG (L R OP)
          (SETQ OP (CADR F))
          (RETURN (COND
                    [(MEMB OP BIFTAG_INFIX)
                      (COND
                        ((AND (NOT (EQ (TYPE ARG)
                                       (QUOTE Erel)))
                              (NOT (EQ (LENGTH ARG) 3)))
                          (ERROR_HANDLER
                            (QUOTE BAD_ARGS) ARG))
                        (T (SETQ L (SEL ARG 1))
                           (SETQ R (SEL ARG 2))
                           (COND
                             [(EQ OP (QUOTE reduction))
                               (COND
                                 ((EQ (TYPE L)
                                      (QUOTE closure))
                                   (LIST (QUOTE closure)
                                         (QUOTE ?A)
                                         (LIST (QUOTE reduce)
                                               (QUOTE ?A)
                                               (QUOTE by)
                                               L
                                               (QUOTE from)
                                               R) ENV-I))
```

```
      (T (ERROR_HANDLER
           (QUOTE EXP_FUNC)
           (CAR ARG]
[(MEMB OP NUMOP)
  (COND
    [(AND (NUMBERP L)
          (NUMBERP R))
      (COND
        ((EQ OP (QUOTE +))
          (PLUS L R))
        ((EQ OP (QUOTE -))
          (DIFFERENCE L R))
        ((EQ OP (QUOTE *))
          (TIMES L R))
        [(EQ OP (QUOTE /))
          (COND
            ((ZEROP R)
              (ERROR_HANDLER
                (QUOTE ZERO_DIV)
                (CADDR IEXP)))
            (T (QUOTIENT L R]
        ((EQ OP (QUOTE <))
          (TF (LESSP L R)))
        ((EQ OP (QUOTE >))
          (TF (GREATERP L R)))
        ((EQ OP (QUOTE <=))
          (TF (LEQ L R)))
        ((EQ OP (QUOTE >=))
          (TF (GEQ L R)))
        (T (QUOTE impossible]
    (T (ERROR_HANDLER
         (QUOTE EXP_NUM)
         (LIST (CAR IEXP)
               (QUOTE or)
               (CADDR IEXP]
[(EQ OP (QUOTE or))
  (COND
    ((OR (EQ L (QUOTE true))
         (EQ R (QUOTE true)))
      (QUOTE true))
    (T (QUOTE false]
[(EQ OP (QUOTE and))
  (COND
    ((AND (EQ L (QUOTE true))
          (EQ R (QUOTE true)))
      (QUOTE true))
    (T (QUOTE false]
[(MEMB OP SETOP)
  (COND
    [[AND (MEMB (TYPE L)
                (QUOTE (Eset Erel)))
```

```
    (MEMB (TYPE R)
      (QUOTE (Eset Erel]
(COND
  [(EQ OP (QUOTE union))
    (COND
      [(OR (EQ (CAR L)
               (QUOTE Eset))
           (EQ (CAR R)
               (QUOTE Eset)))
        (CONS (QUOTE Eset)
              (UNION (CDR L)
                     (CDR R]
      (T (CONS (QUOTE Erel)
               (UNION (CDR L)
                      (CDR R]
  [(EQ OP (QUOTE intersection))
    (COND
      [(OR (EQ (CAR L)
               (QUOTE Erel))
           (EQ (CAR R)
               (QUOTE Erel)))
        (CONS (QUOTE Erel)
         (INTERSECTION (CDR L)
                       (CDR R]
      (T (CONS (QUOTE Eset)
          (INTERSECTION (CDR L)
                        (CDR R]
  [(EQ OP (QUOTE setdiff))
    (COND
      [(EQ (CAR L)
           (QUOTE Eset))
        (CONS (QUOTE Eset)
         (LDIFFERENCE (CDR L)
                      (CDR R]
      (T (CONS (QUOTE Erel)
          (LDIFFERENCE (CDR L)
                       (CDR R]
  [(EQ OP (QUOTE cart))
    (CONS (QUOTE Erel)
          (CART_PROD
            (CDR L)
            (CDR R]
  [(EQ OP (QUOTE !subset))
    (COND
      ((GREATERP (LENGTH L)
                 (LENGTH R))
        (QUOTE false))
      (T (DO_SUBSET
           (CDR L)
           (CDR R]
```

```
        [(EQ OP (QUOTE subset))
          (COND
            ((GEQ (LENGTH L)
                  (LENGTH R))
              (QUOTE false))
            (T (DO_SUBSET
                 (CDR L)
                 (CDR R]
        (T (QUOTE impossible]
    (T (ERROR_HANDLER
         (QUOTE EXP_SET)
         (LIST (CAR IEXP)
               (QUOTE or)
               (CADDR IEXP]
((EQ OP (QUOTE =))
  (REQUAL L R ENV-I))
((EQ OP (QUOTE <>))
  (RNOT (REQUAL L R ENV-I)))
((EQ OP (QUOTE member))
  (MEM L R))
((EQ OP (QUOTE nomem))
  (RNOT (MEM L R)))
[(MEMB OP
  (QUOTE (%! construction
          concatenation)))
  (COERCE_TO_REL L)
  (COERCE_TO_REL R)
  (COND
    ((EQ ERRORCODE (QUOTE ERRORFREE))
      (COND
        [(EQ OP (QUOTE %!))
          (CONS (QUOTE Erel)
                (RELATIVE_PRODUCT
                  (CDR L)
                  (CDR R]
        ((EQ OP (QUOTE construction))
          (CONSTRUCTION L R))
        ((EQ OP (QUOTE concatenation))
          (ARRAY_CONCATENATION
            L R]
[(MEMB OP (QUOTE (sel seq_to_array
                  select_all rel_prod)))
  (COERCE_TO_REL L)
  (COND
    ((EQ ERRORCODE (QUOTE ERRORFREE))
      (COND
        ((EQ OP (QUOTE sel))
          (SEL L R))
        ((EQ OP (QUOTE seq_to_array))
          (SEQ_TO_ARRAY L R))
```

```
          ((EQ OP (QUOTE select_all))
            (SELECT_ALL R (CDR L)))
          ((EQ OP (QUOTE rel_prod))
            (COND
              [(EQ (TYPE R)
                   (QUOTE closure))
                (CONS (QUOTE Erel)
                      (MAPRP R (CDR L]
              (T (ERROR_HANDLER
                   (QUOTE EXP_FUNC) R]
((EQ OP (QUOTE :))
  (FORM_PAIR L R))
[(EQ OP (QUOTE img))
  (COND
    ([AND (EQ (TYPE L)
              (QUOTE closure))
          (MEMB (TYPE R)
                (QUOTE (Eset Erel]
      (CONS (QUOTE Eset)
            (MAPIMG L (CDR R)
                    ENV-I)))
    (T (COND
         ((EQ (TYPE L)
              (QUOTE closure))
          (ERROR_HANDLER
            (QUOTE EXP_SET)
            (CADDR IEXP)))
         (T (ERROR_HANDLER
              (QUOTE EXP_FUNC)
              (CAR IEXP]
[(MEMB OP (QUOTE (rel_prod_inv
                  isomorphism)))
  (COERCE_TO_REL R)
  (COND
    [(AND (EQ (TYPE L)
              (QUOTE closure))
          (EQ ERRORCODE
              (QUOTE ERRORFREE)))
      (COND
        [(EQ OP (QUOTE rel_prod_inv))
          (CONS (QUOTE Erel)
                (MAPRP_INV L
                  (CDR R]
        ((EQ OP (QUOTE isomorphism))
          (CONS (QUOTE Erel)
                (MAP_ISOMORPHISM
                  L (CDR R]
    (T (COND
         ((NOT (EQ (TYPE L)
                   (QUOTE closure)))
```

```
                                (ERROR_HANDLER
                                  (QUOTE EXP_FUNC) L]
                          [(EQ OP (QUOTE filter))
                            (COND
                              ([AND (EQ (TYPE L)
                                        (QUOTE closure))
                                    (MEMB (TYPE R)
                                          (QUOTE (Eset Erel]
                                 (CONS (CAR R)
                                       (FILTER L (CDR R)
                                                 ENV-I)))
                              (T (COND
                                   ((EQ (TYPE L)
                                        (QUOTE closure))
                                    (ERROR_HANDLER
                                      (QUOTE EXP_SET)
                                      (CADDR IEXP)))
                                   (T (ERROR_HANDLER
                                        (QUOTE EXP_BOOL)
                                        (CAR IEXP]
                          ((EQ OP (QUOTE superscript))
                            (SUPERSCRIPT L R]
                (T (COND
                     ((EQ OP (QUOTE not))
                       (RNOT ARG))
                     ((EQ OP (QUOTE file))
                       (FILE_READ ARG))
                     [(EQ OP (QUOTE unitset))
                       (COND
                         (OR (ATOM ARG) (STRINGP ARG)
                           (LIST (QUOTE Eset) ARG))
                         (T (COND
                              ((MEMB (CAR ARG)
                                 (QUOTE (Eset Erel closure)))
                                (LIST (QUOTE Eset) ARG))
                              (T (LIST (QUOTE Erel) ARG]
                     [(MEMB OP (QUOTE (unitset_select
                                       elementselect)))
                       (COND
                         [(MEMB (TYPE ARG)
                                (QUOTE (Eset Erel)))
                           (COND
                             [(EQ OP (QUOTE unitset_select))
                               (COND
                                 ((EQ (LENGTH (CDR ARG))
                                      1)
                                   (CADR ARG))
                                 (T (ERROR_HANDLER
                                      (QUOTE EXP_UNITSET)
                                      (CADR PEXP]
```

```
                ((EQ OP (QUOTE elementselect))
                  (COND
                    ((NULL (CDR ARG))
                      (ERROR_HANDLER
                        (QUOTE EXP_NEMPTY)
                        (CADR PEXP)))
                    (T (CADR ARG]
              (T (ERROR_HANDLER
                   (QUOTE EXP_SET)
                   (CADR PEXP]
        [(EQ OP (QUOTE cardinality))
          (COND
            ((MEMB (TYPE ARG)
                   (QUOTE (Eset Erel)))
              (LENGTH (CDR ARG)))
            (T (ERROR_HANDLER
                 (QUOTE EXP_SET)
                 (CADR PEXP]
        [(MEMB OP (QUOTE (converse rng dom
                          array_to_seq run
                          reverse_array
                          curry_ext
                          uncurry_ext)))
          (COERCE_TO_REL ARG)
          (COND
            ((EQ ERRORCODE (QUOTE ERRORFREE))
              (COND
                ((EQ OP (QUOTE converse))
                  (CONVERSE ARG))
                ((EQ OP (QUOTE dom))
                  (DOMAIN ARG))
                ((EQ OP (QUOTE rng))
                  (RANGE ARG))
                [(EQ OP (QUOTE run))
                  (COND
                    ([EQ [LENGTH (MAKE_UNIQUE
                                   (CDR (RANGE ARG)
                                   NIL ENV-P]
                         (LENGTH (CDR (RANGE ARG]
                      (QUOTE true))
                    (T (QUOTE false]
                ((EQ OP (QUOTE reverse_array))
                  (REVERSE_ARRAY ARG))
                ((EQ OP (QUOTE array_to_seq))
                  (ARRAY_TO_SEQ ARG))
                ((EQ OP (QUOTE curry_ext))
                  (CURRY_EXT ARG))
                ((EQ OP (QUOTE uncurry_ext))
                  (UNCURRY_EXT ARG]
        ((EQ OP (QUOTE Hd))
          (HEAD ARG))
```

```
                        ((EQ OP (QUOTE T1))
                          (TAIL ARG))
                        [(MEMB OP (QUOTE (min max unique_set)))
                          (COND
                            [(EQ (TYPE ARG)
                                 (QUOTE Eset))
                              (COND
                                ((EQ OP (QUOTE min))
                                  (MIN_SET ARG))
                                ((EQ OP (QUOTE max))
                                  (MAX_SET ARG)
                                ((EQ OP (QUOTE unique_set))
                                  (MAKE_UNIQUE (CDR ARG)
                                               NIL ENV-P]
                            (T (ERROR_HANDLER
                                 (QUOTE EXP_SET) ARG]
                        ((MEMB OP (QUOTE (rsort sort)))
                          (COND
                            [(NUMERIC_SET (CDR ARG))
                              (COND
                                [(EQ OP (QUOTE rsort))
                                  (CONS (QUOTE Erel)
                                        (SEQ_TO_REL
                                          (SORT (CDR ARG)
                                            (QUOTE LESSP]
                                (T (CONS (QUOTE Erel)
                                         (LIST_TO_REL
                                           (SORT (CDR ARG)
                                             (QUOTE LESSP]
                            (T (ERROR_HANDLER
                                 (QUOTE EXP_NSET) ARG])


(ARRAY_CONCATENATION
**********************************************************************
  args:       A1, A2  (Tagged relations)
  calls:      NUMERIC_SET, DOMAIN, REVERSE, APPEND, MAPCAR,
              PLUS, CONS, ERROR_HANDLER, LIST
  called by:  BIF_APPLY
  binds:      INDEX, X
  comments:   Given two arrays (relation with numeric index),
              A1 and A2, returns a single array which is the
              concatenation of A1 to A2.
**********************************************************************
  [LAMBDA (A1 A2)
    (COND
      [[AND (NUMERIC_SET (CDR (DOMAIN A1)))
             (NUMERIC_SET (CDR (DOMAIN A2]
       (PROG (INDEX)
             (SETQ INDEX (CAAR (REVERSE A1)))
```

```
            (RETURN (APPEND A1 (MAPCAR (CDR A2)
              (QUOTE (LAMBDA (X) (SETQ INDEX (PLUS 1 INDEX))
                                 (CONS INDEX (CDR X]
      (T (ERROR_HANDLER
          (QUOTE EXP_ARRAY)
          (LIST A1 (QUOTE or) A2]

(ARRAY_REDUCTION
*****************************************************************
  args:       EXP, EA
  calls:      COERCE_TO_REL, MAPCAR, ERROR_HANDLER, EV,
              RANGE, RPAPPLY, LIST, CONS
  called by:  EV_SPECIAL_CASES
  binds:      ARRAY, FNC, START, ARGS, ANS, X
  comments:   Given an expression, EXP, of the form:
               "reduce Array by Function from Startin_Point"
              created by the "red" operator, returns a value
              by extracting the function, starting point and
              array, to reduce the values in the array by
              repeated applications of the function.
*****************************************************************
  [LAMBDA (EXP EA)
    (PROG (ARRAY FNC START ARGS ANS)
          (SETQ ARRAY (EV (CADR EXP) EA))
          (COND
           ((COERCE_TO_REL ARRAY)
             (SETQ FNC (CADDDR EXP))
             (SETQ START (CADDDR (CDDR EXP)))
             (SETQ ARGS (CDR (RANGE ARRAY)))
             (SETQ ANS START)
             [MAPCAR ARGS (QUOTE (LAMBDA (X)
               (SETQ ANS (RPAPPLY FNC (LIST (QUOTE Erel)
                                            (CONS 1 ANS)
                                            (CONS 2 X]
             (RETURN ANS))
            (T (ERROR_HANDLER (QUOTE EXP_REL) ARRAY])


(ARRAY_TO_SEQ
*****************************************************************
  args:       ARRAY (Tagged relation)
  calls:      SET, RANGE, CONS, REVERSE
  called by:  BIF_APPLY
  binds:      S1, S2, SEQ
  comments:   Converts the values of an array into a sequence
*****************************************************************
  [LAMBDA (ARRAY)
    (PROG (S1 S2 SEQ)
          (SETQ S1 (CDR (RANGE ARRAY)))
          (SETQ S2 (CDR S1))
          (SET SEQ NIL)
```

```
      LOOP(COND
           [(NULL S2)
             (RETURN (CONS (QUOTE Erel) (REVERSE SEQ]
           (T (SETQ SEQ (CONS (CONS (CAR S1) (CAR S2)) SEQ))
              (SETQ S1 (CDR S1))
              (SETQ S2 (CDR S2))
              (GO LOOP])


(CART_PROD
*****************************************************************
  args:       A, B (Untagged sets)
  calls:      APPEND, MAPCAR, CONS, CART_PROD
  called by:  BIF_APPLY, CART_PROD
  binds:      X
*****************************************************************
  [LAMBDA (A B)
    (COND
      ((NULL A) NIL)
      (T (APPEND [MAPCAR B (QUOTE (LAMBDA (X)
                                     (CONS (CAR A) X]
                 (CART_PROD (CDR A) B])


(CONSTRUCTION
*****************************************************************
  args:       TBL1, TBL2 (Tagged relations)
  calls:      CONS, MAPCAR, INTERSECTION, DOMAIN, LIST, SEL
  called by:  BIF_APPLY
  binds:      X
  comments:   Given two tables, returns a table which relates
              every common domain element of TBL1 and TBL2 to
              a list containing the range element from each
              table, associated with the domain element.
*****************************************************************
  [LAMBDA (TBL1 TBL2)
    (CONS (QUOTE Erel)
          (MAPCAR (CDR (INTERSECTION (DOMAIN TBL1)
                                     (DOMAIN TBL2)))
           (QUOTE (LAMBDA (X) (CONS X (LIST (QUOTE Erel)
                                    (CONS 1 (SEL TBL1 X))
                                    (CONS 2 (SEL TBL2 X])
```

```
(CONVERSE
**********************************************************************
  args:       R (Tagged relation)
  calls:      CONS, REVERSE, MAPCAR
  called by:  BIF_APPLY, SUPERSCRIPT
  binds:      X
  comments:   Given a relation, R, returns a relation with
              the range and domain inverted.
**********************************************************************
  [LAMBDA (R)
    (CONS (CAR R)
          (REVERSE (MAPCAR (CDR R)
                           (QUOTE (LAMBDA (X)
                                          (CONS (CDR X)
                                                (CAR X])

(CURRY_EXT
**********************************************************************
  args:       TBL (Tagged relation)
  calls:      BINARY_LIST, MAPCAR, CONS, REVERSE, LOOKUP,
              CURRY_ELEMENT, LDIFFERENCE
  called by:  BIF_APPLY
  binds:      TAG, PTBL, CTBL, FIRST, KEY, SUBTBL, X
  comments:   Given a table which represents an extensional
              uncurried function, i.e., every domain element
              is a binary list and every range element is the
              result of the represented function on the argu-
              ments in the list, returns a table which repre-
              sents the curried version of the original TBL.
**********************************************************************
  [LAMBDA (TBL)
    (PROG (PTBL TAG FIRST KEY SUBTBL CTBL)
          (SETQ TAG (CAR TBL))
          (SETQ PTBL (CDR TBL))
          (SETQ CTBL NIL)
      LOOP[COND
           [(NULL PTBL)
             (RETURN (CONS TAG (REVERSE CTBL]
           (T (SETQ FIRST (CAR PTBL))
              (COND
                ((BINARY_LIST
                    (CAR FIRST))
                 (SETQ KEY (LOOKUP 1 (CAR FIRST)))
                 (SETQ SUBTBL NIL)
                 [MAPCAR PTBL (QUOTE (LAMBDA (X)
                    (COND
                      [(BINARY_LIST (CAR X))
                        (COND
                          ((EQ (CDADAR X) KEY)
                             (SETQ SUBTBL (CONS X SUBTBL]
                          (T (GO EXIT]
```

```
                    (SETQ CTBL (CONS (CURRY_ELEMENT
                                       KEY SUBTBL) CTBL))
                    (SETQ PTBL (LDIFFERENCE PTBL SUBTBL)))
                   (T (GO EXIT]
            (GO LOOP)
       EXIT])

(DOMAIN
*****************************************************************
  args:       R (Tagged relation)
  calls:      CONS, MAPCAR, CAR
  called by:  BIF_APPLY, ARRAY_CONCATENATION, CONSTRUCTION,
              REFLEXIVE_TRANSITIVE_CLOSURE, REVERSE_ARRAY,
              SEQ_TO_ARRAY
  comments:   Returns a tagged set of the left members of the
              relation, R.
*****************************************************************
  [LAMBDA (R)
    (CONS (QUOTE Eset) (MAPCAR (CDR R) (QUOTE CAR]

(DO_SUBSET
*****************************************************************
  args:       S1, S2 (Untagged relation or set)
  calls:      MEMBER, DO_SUBSET
  called by:  BIF_APPLY, DO_SUBSET, REQUAL
*****************************************************************
  [LAMBDA (S1 S2)
    (COND
      ((NULL S1) (QUOTE true))
      ((MEMBER (CAR S1) S2)
       (DO_SUBSET (CDR S1) S2))
      (T (QUOTE false])

(FILE_READ
*****************************************************************
  args:       FNAME (Unix filename)
  calls:      WRITE, INFILE, TERPRI, CLOSEALL, MKATOM,
              INFILEP, READ
  called by:  BIF_APPLY
  binds:      INPUT
  comments:   Reads from a file, a previously stored RPL data
              element.
*****************************************************************
  [LAMBDA (FNAME)
    (SETQ FNAME (MKATOM FNAME))
    (PROG (INPUT)
          (SETQ INPUT (INFILEP FNAME))
          (COND
           ((NULL INPUT)
             (WRITE (QUOTE (file not found)))
             (GO EXIT)))
```

```
          (INFILE INPUT)
          (RETURN (READ INPUT))
      EXIT(TERPRI)
          (CLOSEALL NIL])

(FILE_WRITE
**************************************************************
  args:       FNAME (Unix filename)
              EXP   (Any RPL expression)
  calls:      OUTFILE, PRINT, CLOSEALL, MKATOM, OUTFILEP
  called by:  EXECUTE
  binds:      OUTPUT
  comments:   Writes the evaluated EXP to the file FNAME.
**************************************************************
  [LAMBDA (FNAME EXP)
    (SETQ FNAME (MKATOM FNAME))
    (PROG (OUTPUT)
          (SETQ OUTPUT (OUTFILEP FNAME))
          (OUTFILE OUTPUT)
          (PRINT EXP OUTPUT)
          (CLOSEALL NIL])

(FILTER
**************************************************************
  args:       P (RPL boolean predicate, evaluated)
              S (Untagged set or relation)
  calls:      MAPCAR, RPAPPLY, CONS, REVERSE
  called by:  BIF_APPLY
  binds:      FSET, X, ARG
  uses free:  ERRORCODE, FILTER_ON
  comments:   Returns S or a subset of S, based upon the
              result of applying the boolean predicate, P, to
              each element of S.
**************************************************************
  [LAMBDA (P S)
    (PROG (FSET ARG)
          (SETQ FSET NIL)
          (SETQ FILTER_ON T)
          [MAPCAR S (QUOTE (LAMBDA (X)
               [COND
                ((EQ (RPAPPLY P X) (QUOTE true))
                   (SETQ FSET (CONS X FSET]
                   (SETQ ERRORCODE (QUOTE ERRORFREE]
          (SETQ FILTER_ON NIL)
          (RETURN (REVERSE FSET])
```

```
(FNC_BODY
**************************************************************
  args:       N
  calls:      LIST, DIFFERENCE
  called by:  REPEAT_COMPOSITION
  binds:      ANS
  comments:   An auxiliary function to REPEAT_COMPOSITION
              which creates the physical closure with N
              compositions of a function f.
**************************************************************
  [LAMBDA (N)
    (PROG (ANS)
          (SETQ ANS (LIST (QUOTE f) (QUOTE x)))
      LOOP(COND
           ((EQ N 0) (RETURN ANS))
           (T (SETQ ANS (LIST (QUOTE f) ANS))
              (SETQ N (DIFFERENCE N 1))
              (GO LOOP])

(FORM_PAIR
**************************************************************
  args:       X (An elementary pair)
  calls:      ERROR_HANDLER
  called by:  BIF_APPLY
**************************************************************
  [LAMBDA (X Y) (CONS X Y])

(HEAD
**************************************************************
  args:       X, Y (Anything)
  calls:      CONS
  called by:  BIF_APPLY
**************************************************************
  [LAMBDA (X)
    (COND
      ((AND (LISTP X) (NOT (NULL X))) (CAR X))
      (T (ERROR_HANDLER (QUOTE EXP_PAIR) X])

(MAPIMG
**************************************************************
  args:       F (RPL function, evaluated form)
              S (Untagged set of relation)
  calls:      MAPCAR, RPAPPLY
  called by:  BIF_APPLY
  binds:      X
  comments:   Returns an untagged set of results of applying
              F to each member of S.
**************************************************************
  [LAMBDA (F S)
    (MAPCAR S (QUOTE (LAMBDA (X) (RPAPPLY F X])
```

```
(MAPRP
*****************************************************************
  args:       F   (RPL function, evaluated form)
              TBL (Untagged relation)
  calls:      MAPCAR, CONS, RPAPPLY
  called by:  BIF_APPLY
  binds:      X
  comments:   Returns an untagged table which relates each
              domain element of TBL to the result of applying
              F to the associated range element.
*****************************************************************
  [LAMBDA (F TBL)
    (MAPCAR TBL (QUOTE (LAMBDA (X)
                               (CONS (CAR X)
                                     (RPAPPLY F (CDR X])

(MAPRP_INV
*****************************************************************
  args:       F   (RPL function, evaluated form)
              TBL (Untagged relation)
  calls:      MAPCAR, CONS, RPAPPLY
  called by:  BIF_APPLY
  binds:      X
  comments:   Returns an untagged table which applys F to
              each domain element of TBL, and relates this
              result to the associated range element.
*****************************************************************
  [LAMBDA (F TBL)
    (MAPCAR TBL (QUOTE (LAMBDA (X)
                               (CONS (RPAPPLY F (CAR X))
                                     (CDR X])

(MAP_ISOMORPHISM
*****************************************************************
  args:       F   (RPL function, evaluated form)
              TBL (Untagged relation)
  calls:      MAPCAR, CONS, RPAPPLY
  called by:  BIF_APPLY
  binds:      X
  comments:   Returns an untagged table where each element is
              the result of applying F to both the left and
              right member of each element in TBL.
*****************************************************************
  [LAMBDA (F TBL)
    (MAPCAR TBL (QUOTE (LAMBDA (X)
                               (CONS (RPAPPLY F (CAR X))
                                     (RPAPPLY F (CDR X])
```

```
(MAX_SET
**********************************************************************
  args:       S (Tagged numeric set)
  calls:      NUMERIC_SET, GREATERP, MAPCAR, T, ERROR_HANDLER
  called by:  BIF_APPLY
  binds:      SET, MAX, X
  comments:   Returns the maximum member of the set.
**********************************************************************
  [LAMBDA (S)
    (PROG (MAX SET)
          (SETQ SET (CDR S))
          (COND
           ((NUMERIC_SET SET)
             (SETQ MAX (CAR SET))
             [MAPCAR SET (QUOTE (LAMBDA (X)
                                        (COND
                                          ((GREATERP X MAX)
                                            (SETQ MAX X]
             (RETURN MAX)))
          (T (ERROR_HANDLER (QUOTE EXP_NSET) SET])

(MEM
**********************************************************************
  args:       X (Anything)
              S (A tagged set or relation)
  calls:      TYPE, MEMBER, ERROR_HANDLER
  called by:  BIF_APPLY
  comments:   Returns true if X is a member of S, otherwise
              false is returned.
**********************************************************************
  [LAMBDA (X S)
    (COND
      [(MEMBER (TYPE S) (QUOTE (Eset Erel)))
       (COND
         ((EQ (MEMBER X S) NIL) (QUOTE false))
         (T (QUOTE true]
      (T (ERROR_HANDLER (QUOTE EXP_SET) S])


(MIN_SET
**********************************************************************
  args:       S (Tagged numeric set)
  calls:      NUMERIC_SET, LESSP, MAPCAR, ERROR_HANDLER
  called by:  BIF_APPLY
  binds:      SET, MIN, X
  comments:   Returns the minimum member of the set.
**********************************************************************
  [LAMBDA (S)
    (PROG (MIN SET)
          (SETQ SET (CDR S))
```

```
          (COND
           ((NUMERIC_SET SET)
             (SETQ MIN (CAR SET))
             [MAPCAR SET (QUOTE (LAMBDA (X)
                                        (COND
                                          ((LESSP X MIN)
                                            (SETQ MIN X]
             (RETURN MIN))
           (T (ERROR_HANDLER (QUOTE EXP_NSET) SET])

(RANGE
*****************************************************************
  args:       R (Tagged relation)
  calls:      CONS, MAPCAR, CDR
  called by:  BIF_APPLY, ARRAY_REDUCTION, ARRAY_TO_SEQ,
              REFLEXIVE_TRANSITIVE_CLOSURE, SEQ_TO_ARRAY
  comments:   Returns a tagged set consisting of the right
              members of the relation, R.
*****************************************************************
  [LAMBDA (R)
    (CONS (QUOTE Eset) (MAPCAR (CDR R) (QUOTE CDR])

(REFLEXIVE_TRANSITIVE_CLOSURE
*****************************************************************
  args:       R (Tagged relation)
  calls:      UNION, DOMAIN, RANGE, CONS, MAPCAR,
              TRANSITIVE_CLOSURE
  called by:  SUPERSCRIPT
  binds:      TAG, MEM, X
*****************************************************************
  [LAMBDA (R)
    (PROG (TAG MEM)
          (SETQ TAG (CAR R))
          [SETQ MEM (UNION (CDR (DOMAIN R)) (CDR (RANGE R]
          (RETURN (CONS TAG
            (UNION [MAPCAR MEM (QUOTE (LAMBDA (X) (CONS X X]
                   (CDR (TRANSITIVE_CLOSURE R])
```

```
(RELATIVE_PRODUCT
**********************************************************************
  args:       TBL1, TBL2 (Untagged relations)
  calls:      APPEND, MAPCAR, SELECT_ALL, CONS, RELATIVE_PRODUCT
  called by:  BIF_APPLY, RELATIVE_PRODUCT, TRANSITIVE_CLOSURE
  binds:      X
  comments:   Returns an untagged table which takes the right
              member of each element in TBL1 and relates it
              to the set of all right members it is related
              to in TBL2.
**********************************************************************
  [LAMBDA (TBL1 TBL2)
    (COND
      ((NULL TBL1) NIL)
      (T (APPEND [MAPCAR (CDR (SELECT_ALL
                                 (CDAR TBL1) TBL2))
                         (QUOTE (LAMBDA (X)
                                        (CONS (CAAR TBL1) X]
                 (RELATIVE_PRODUCT (CDR TBL1) TBL2])

(REPEAT_COMPOSITION
**********************************************************************
  args:       FNC (RPL function, evaluated form)
              P   (A positive integer)
  calls:      CONS, LIST, FNC_BODY
  called by:  SUPERSCRIPT
  binds:      SE
  comments:   A special case of the "sup" command.  Given a
              function, FNC, and the number of times, P, FNC
              is to be composed with itself, returns a
              closure which represents the resulting function
**********************************************************************
  [LAMBDA (FNC P)
    (PROG (SE)
          (SETQ SE (CONS (CONS (QUOTE f) FNC)
                         (CADDDR FNC)))
          (RETURN (LIST (QUOTE closure)
                        (QUOTE x)
                        (FNC_BODY P) SE])


(REQUAL
**********************************************************************
  args:       X, Y (Anything)
  calls:      MEMBER, TYPE, DO_SUBSET, TF, EQUAL
  called by:  BIF_APPLY
**********************************************************************
  [LAMBDA (X Y)
    (COND
      [[AND (MEMBER (TYPE X) (QUOTE (Eset Erel)))
            (MEMBER (TYPE Y) (QUOTE (Eset Erel]
```

```
        (COND
          ((AND (EQ (DO_SUBSET (CDR X) (CDR Y)) (QUOTE true))
                (EQ (DO_SUBSET (CDR Y) (CDR X)) (QUOTE true)))
            (QUOTE true))
          (T (QUOTE false]
       (T (TF (EQUAL X Y])


(REVERSE_ARRAY
**********************************************************************
  args:       LST (Tagged relation)
  calls:      SORT, DOMAIN, PLUS, REVERSE, CONS, MAPCAR,
              DIFFERENCE, LESSP
  called by:  BIF_APPLY
  binds:      TAG, DOM, K, X
  comments:   Given an array, LST, returns an array with the
              values in reverse order.
**********************************************************************
  [LAMBDA (LST)
    (PROG (TAG DOM K)
          (SETQ TAG (CAR LST))
          (SETQ DOM (SORT (CDR (DOMAIN LST)) (QUOTE LESSP)))
          (SETQ K (PLUS (CAR (REVERSE DOM)) (CAR DOM)))
          (RETURN (CONS TAG
            (REVERSE (MAPCAR (CDR LST)
               (QUOTE (LAMBDA (X) (CONS (DIFFERENCE K (CAR X))
                                        (CDR X])


(RNOT
**********************************************************************
  args:       B (LISP boolean)
  called by:  BIF_APPLY
  comments:   RPL negation
**********************************************************************
  [LAMBDA (B)
    (COND
      ((EQ B (QUOTE true)) (QUOTE false))
      (T (QUOTE true])
```

```
(RPL_REPEAT
**********************************************************************
  args:       EXP, ER
  calls:      ERROR_HANDLER, EV, TYPE, RPAPPLY
  called by:  EV_SPECIAL_CASES
  binds:      F, P, X, RESULT
  uses free:  ERRORCODE
  comments:   Given an expression of the form:
               "repeat (F X) until_not P"
              created by the "iter" operation, continues to
              apply F to X until the predicate P is true.
**********************************************************************
  [LAMBDA (EXP ER)
    (PROG (F P X RESULT)
          (SETQ F (EV (CAADR EXP) ER))
          (SETQ P (EV (CAADDR (CDR EXP)) ER))
          (SETQ X (EV (QUOTE ?x) ER))
          (COND
           ([NOT (AND (EQ ERRORCODE (QUOTE ERRORFREE))
                      (EQ (TYPE F) (QUOTE closure))
                      (EQ (TYPE P) (QUOTE closure]
             (ERROR_HANDLER (QUOTE EXP_FUNC)
               (QUOTE (boolean predicate missing or
                       bad function definition in iter)))
             (GO EXIT)))
          (SETQ RESULT (RPAPPLY F X))
      LOOP(COND
           ((EQ (RPAPPLY P RESULT) (QUOTE true))
             (SETQ RESULT (RPAPPLY F RESULT))
             (GO LOOP)))
          (RETURN RESULT)
      EXIT])

(SEL
**********************************************************************
  args:       TBL (Tagged relation)
              TGT (Anything)
  calls:      SASSOC, ERROR_HANDLER, LIST
  called by:  BIF_APPLY, CONSTRUCTION
  binds:      X
  comments:   Returns the right member of the first
              occurrence of X as a left member.
**********************************************************************
  [LAMBDA (TBL TGT)
    (PROG (X)
          (SETQ X (SASSOC TGT (CDR TBL)))
          (RETURN (COND
                    ((EQ X NIL) (ERROR_HANDLER (QUOTE UBTE)
                                  (LIST TGT (QUOTE (not found in))
                                        TBL)))
                    (T (CDR X])
```

```
(SEQ_TO_ARRAY
**************************************************************
  args:       SEQ   (Tagged relation)
              INDEX (A positive integer)
  calls:      LDIFFERENCE, DOMAIN, RANGE, LIST, CONS, LOOKUP,
              REVERSE, PLUS
  called by:  BIF_APPLY
  binds:      FIRST, ARRAY
  comments:   Converts a sequence to an array indexed from
              INDEX.
**************************************************************
  [LAMBDA (SEQ INDEX)
    (PROG (FIRST ARRAY)
          [SETQ FIRST (CAR (LDIFFERENCE (DOMAIN SEQ)
                                        (RANGE SEQ]
          (SETQ ARRAY (LIST (CONS INDEX FIRST)))
      LOOP(SETQ FIRST (LOOKUP FIRST (CDR SEQ)))
          (COND
           [(EQ FIRST NIL)
             (RETURN (CONS (QUOTE Erel) (REVERSE ARRAY]
           (T (SETQ INDEX (PLUS 1 INDEX))
              (SETQ ARRAY (CONS (CONS INDEX FIRST) ARRAY))
              (GO LOOP])

(SUPERSCRIPT
**************************************************************
  args:       OPND (Tagged relation or RPL function)
              PWR  (+, **, or a positive integer)
  calls:      EQUAL, NUMBERP, GREATERP, TYPE,
              REFLEXIVE_TRANSITIVE_CLOSURE, CONVERSE,
              TRANSITIVE_CLOSURE, REPEAT_COMPOSITION,
              ERROR_HANDLER
  called by:  BIF_APPLY
  uses free:  IEXP
  comments:   Handles all cases of the operator "sup".
**************************************************************
  [LAMBDA (OPND PWR)
    (COND
      ((AND (EQUAL PWR (QUOTE (closure +)))
            (EQ (TYPE OPND) (QUOTE Erel)))
       (TRANSITIVE_CLOSURE OPND))
      ((AND (EQUAL PWR (QUOTE (closure star)))
            (EQ (TYPE OPND) (QUOTE Erel)))
       (REFLEXIVE_TRANSITIVE_CLOSURE OPND))
      ((AND (NUMBERP PWR) (EQ PWR -1)
            (EQ (TYPE OPND) (QUOTE Erel)))
       (CONVERSE OPND))
      ((AND (NUMBERP PWR) (GREATERP PWR 0)
            (EQ (TYPE OPND) (QUOTE closure)))
       (REPEAT_COMPOSITION OPND PWR))
      (T (ERROR_HANDLER (QUOTE BAD_SYNTAX) IEXP])
```

```
(TAIL
**********************************************************************
  args:       X (An elementary pair)
  calls:      ERROR_HANDLER
  called by:  BIF_APPLY
**********************************************************************
  [LAMBDA (X)
    (COND
      ((AND (LISTP X) (NOT (NULL X))) (CDR X))
      (T (ERROR_HANDLER (QUOTE EXP_PAIR) X])

(TRANSITIVE_CLOSURE
**********************************************************************
  args:       R (Tagged relation)
  calls:      CONS, RELATIVE_PRODUCT, UNION
  called by:  REFLEXIVE_TRANSITIVE_CLOSURE, SUPERSCRIPT
  binds:      TMP, ANS
**********************************************************************
  [LAMBDA (R)
    (PROG (TMP ANS)
          (SETQ TMP (CDR R))
          (SETQ ANS (CDR R))
      RPLOOP
          (COND
           ((NULL TMP)
             (RETURN (CONS (CAR R) ANS)))
           (T (SETQ TMP (RELATIVE_PRODUCT TMP (CDR R)))
              (SETQ ANS (UNION ANS TMP))
              (GO RPLOOP])

(UNCURRY_EXT
**********************************************************************
  args:       TBL (Tagged relation)
  calls:      COERCE_TO_REL, CONS, APPEND, MAPCAR,
              LIST_TO_REL, LIST
  called by:  BIF_APPLY
  binds:      TAG, PTBL, KEY, SUBTBL, UTBL, X
  comments:   The converse of CURRY_EXT.
**********************************************************************
  [LAMBDA (TBL)
    (PROG (TAG PTBL KEY SUBTBL UTBL)
          (SETQ TAG (CAR TBL))
          (SETQ PTBL (CDR TBL))
      LOOP[COND
           ((NULL PTBL) (RETURN (CONS TAG UTBL)))
           (T (SETQ KEY (CAAR PTBL))
              (SETQ SUBTBL (CDAR PTBL))
              (COND
                ((COERCE_TO_REL SUBTBL)
                  (SETQ SUBTBL (CDR SUBTBL))
```

```
                    [SETQ UTBL (APPEND UTBL
                      (MAPCAR SUBTBL (QUOTE (LAMBDA (X)
                         (CONS [CONS (QUOTE Erel)
                                     (LIST_TO_REL (LIST KEY (CAR X]
                               (CDR X]
                    (SETQ PTBL (CDR PTBL))
                    (GO LOOP]
      EXIT])


(ALL_PAIRS
**********************************************************************
  args:       S (Untagged set)
  calls:      MEMB, ALL_PAIRS
  called by:  EV_SPECIAL_CASES, COERCE_TO_REL, ALL_PAIRS
  comments:   A boolean utility function which determines if
              all the elements of S are elementary pairs.
**********************************************************************
  [LAMBDA (S)
    (COND
      ((NULL S) T)
      ([AND (LISTP (CAR S))
            (NOT (MEMB (CAAR S)
                       (QUOTE (Eset Erel closure]
       (ALL_PAIRS (CDR S])


(BINARY_LIST
**********************************************************************
  args:       REL (Tagged relation)
  calls:      COERCE_TO_REL, ERROR_HANDLER, LIST
  called by:  CURRY_EXT
  comments:   A boolean utility function which verifies that
              REL is an RPL binary list.
**********************************************************************
  [LAMBDA (REL)
    (COND
      ((COERCE_TO_REL REL)
       (COND
         ((AND (EQ (CAADR REL) 1)
                (EQ (CAADDR REL) 2)) T)
         (T (ERROR_HANDLER (QUOTE BAD_ARG)
              (LIST REL (QUOTE (not a binary list])
```

```
(COERCE_TO_REL
**********************************************************************
  args:       S (Tagged relation)
  calls:      ALL_PAIRS, TYPE, ERROR_HANDLER
  called by:  ARRAY_REDUCTION, UNCURRY_EXT, BINARY_LIST,
              BIF_APPLY
  binds:      STYPE
  comments:   A utility function which changes the tags on a
              relation if S is a set and is equivalent to a
              relation.
**********************************************************************
  [LAMBDA (S)
    (PROG (STYPE)
          (SETQ STYPE (TYPE S))
          (RETURN (COND
                    ((EQ STYPE (QUOTE Erel)))
                    ((AND (EQ STYPE (QUOTE Eset))
                          (ALL_PAIRS (CDR S))) S)
                    (T (ERROR_HANDLER (QUOTE EXP_REL) S])


(CURRY_ELEMENT
**********************************************************************
  args:       KEY (Anything)
              TBL (Untagged relation)
  calls:      CONS, REVERSE, MAPCAR, LOOKUP
  called by:  CURRY_EXT
  binds:      X
  comments:   A auxiliary function to CURRY_EXT, which forms
              the curried element, given the KEY and the un-
              curried table, TBL.
**********************************************************************
  [LAMBDA (KEY TBL)
    (CONS KEY (CONS (QUOTE Erel)
                    (REVERSE (MAPCAR TBL (QUOTE (LAMBDA (X)
                             (CONS (LOOKUP 2 (CAR X)) (CDR X])
```

```
(DISPLAY_ENV
**********************************************************************
  args:       FNC (An identifier or nothing)
  calls:      SPACES, WRITE, TERPRI, GET_ENV
  called by:  DISPLAY
  binds:      ENV
  uses free:  USERDEFS
  comments:   Executes the "env" operator.  Displays the
              entire environment if no argument is given or
              the environment associated with the identifier,
              FNC.  The environment is displayed in
              definitional form.
**********************************************************************
  [LAMBDA (FNC)
    (PROG (ENV)
          (SETQ ENV (GET_ENV USERDEFS FNC))
      LOOP(COND
            ((NULL ENV)
              (SPACES 1)
              (WRITE (QUOTE (System Defined Functions)))
              (TERPRI))
            (T (SPACES 1)
               (WRITE (CDAR ENV))
               (TERPRI)
               (SETQ ENV (CDR ENV))
               (GO LOOP])


(GET_ENV
**********************************************************************
  args:       L, FNAME
  calls:      MAPCAR
  called by:  DISPLAY_ENV
  binds:      ENV, X
  comments:   An auxiliary function to DISPLAY_ENV which
              returns only that portions of L (USERDEFS)
              which are in the scope of FNAME.
**********************************************************************
  [LAMBDA (L FNAME)
    (PROG (ENV)
          (COND
           ((NULL FNAME) (RETURN L))
           (T (SETQ ENV L)
              (MAPCAR L (QUOTE (LAMBDA (X)
                (COND
                  ((EQ (CAR X) FNAME) (RETURN ENV))
                  (T (SETQ ENV (CDR ENV])
```

```
(LIST_TO_REL
**********************************************************************
  args:       D (Untagged LISP list)
  calls:      GREATERP, PLUS, LENGTH, DIFFERENCE, CONS,
              MAP2CAR
  called by:  EVSEQ, BIF_APPLY, UNCURRY_EXT
  binds:      R, C
  comments:   Returns an untagged relation which represents
              the appropiate list, D.
**********************************************************************
  [LAMBDA (D)
    (PROG (R C)
          (SETQ R NIL)
          (SETQ C (PLUS (LENGTH D) 1))
      LOOP(SETQ C (DIFFERENCE C 1))
          (COND
           ((GREATERP C 0)
             (SETQ R (CONS C R))
             (GO LOOP))
           (T R))
          (RETURN (MAP2CAR R D (QUOTE CONS])


(LOOKUP
**********************************************************************
  args:       TGT (Anything)
              TBL (Untagged relation)
  calls:      SASSOC
  called by:  DEF_BINDING, DISPLAY, ERROR_HANDLER, EV,
              CURRY_EXT, SEQ_TO_ARRAY, CURRY_ELEMENT
  binds:      X
  comments:   Returns the right member of TBL given the left
              member, TGT, if TGT is found, else returns NIL.
**********************************************************************
  [LAMBDA (TGT TBL)
    (PROG (X)
          (SETQ X (SASSOC TGT TBL))
          (RETURN (COND
                    ((EQ X NIL) NIL)
                    (T (CDR X])
```

```
(MAKE_UNIQUE
*****************************************************************
  args:       INPUT (Untagged set or relation)
              RESULT, ENV
  calls:      MEMBER, REVERSE, EV, MAKE_UNIQUE, CONS
  called by:  EV_SPECIAL_CASES, BIF_APPLY, MAKE_UNIQUE
  comments:   Eliminates redundant elements from INPUT and
              returns RESULT.
*****************************************************************
  [LAMBDA (INPUT RESULT ENV)
    (COND
      ((NULL INPUT) (REVERSE RESULT))
      (T (COND
          ((MEMBER (EV (CAR INPUT) ENV) RESULT)
            (MAKE_UNIQUE (CDR INPUT) RESULT ENV))
          (T (SETQ RESULT (CONS (EV (CAR INPUT) ENV)
                                RESULT))
             (MAKE_UNIQUE (CDR INPUT) RESULT ENV]))


(NUMERIC_SET
*****************************************************************
  args:       SET (Untagged set)
  calls:      NUMBERP, MAPCAR
  called by:  BIF_APPLY, ARRAY_CONCATENATION, MAX_SET, MIN_SET
  binds:      X
  comments:   A boolean utility function which determines if
              all members of SET are numeric.
*****************************************************************
  [LAMBDA (SET)
    (PROG NIL
          [MAPCAR SET (QUOTE (LAMBDA (X)
                                     (COND
                                       ((NOT (NUMBERP X))
                                         (GO EXIT]
          (RETURN T)
      EXIT(RETURN NIL])
```

```
(POSIT
**********************************************************************
  args:       L       (Any LISP list)
              TARGET  (Anything)
  calls:      EQUAL, SET, PLUS
  called by:  EXECUTE
  binds:      N
  comments:   A utility function used to find the position of
              the "==" symbol in an RPL command.
**********************************************************************
  [LAMBDA (L TARGET)
    (PROG (N)
          (SET (QUOTE N) 0)
      LOOP(COND
           ((NULL L) (RETURN 0))
           ((EQUAL TARGET (CAR L))
             (RETURN (PLUS N 1)))
           (T (SET (QUOTE N) (PLUS N 1))
              (SET (QUOTE L) (CDR L))
              (GO LOOP])


(PRINT_LIST
**********************************************************************
  args:       S (Any LISP list)
  calls:      ATOM, STRINGP, MEMB, SHOW_ATOM, PRINT_LIST
  called by:  ERROR_HANDLER, SHOW_ATOM, PRINT_LIST
  comments:   An output utility to display RPL results which
              are in a LISP list form in a more readable
              format.
**********************************************************************
  [LAMBDA (S)
    (COND
      ((NULL S) NIL)
      ([OR (ATOM S)
           (STRINGP S)
           (EQ (CAR S) (QUOTE closure))
           (MEMB (CAR S) (QUOTE (Eset Erel]
       (SHOW_ATOM S))
      (T (SHOW_ATOM (CAR S))
        (PRINT_LIST (CDR S])


(READCMD
**********************************************************************
  calls:      WAITFORINPUT, READLINE
  called by:  RPL
**********************************************************************
  [LAMBDA NIL (WAITFORINPUT) (READLINE])
```

```
(READTERM
**********************************************************************
  calls:      WAITFORINPUT, READLINE
  called by: SET_USER_ENV, EXIT
**********************************************************************
  [LAMBDA NIL (WAITFORINPUT) (CAR (READLINE])


(READ_USER_DEFS
**********************************************************************
  args:       FNAME (Unix filename)
  calls:      WRITE, INFILE, EXECUTE, TERPRI, CLOSEALL,
              INFILEP, READ
  called by: SET_USER_ENV
  binds:      INPUT, DEFIN
  uses free: ERRORCODE
  comments:  A utility function which reads a previous RPL
              session's commands from FNAME into the current
              RPL session.
**********************************************************************
  [LAMBDA (FNAME)
    (PROG (INPUT DEFIN)
          (SETQ INPUT (INFILEP FNAME))
          (COND
           ((NULL INPUT)
             (WRITE (QUOTE (file not found)))
             (GO EXIT)))
          (INFILE INPUT)
          (WRITE (QUOTE (Loading---)))
          (SETQ DEFIN (READ INPUT))
     LOOP(COND
           ((EQ DEFIN (QUOTE EOF))
             (WRITE (QUOTE (Session loaded)))
             (GO EXIT))
           (T (SETQ ERRORCODE (QUOTE ERRORFREE))
              (EXECUTE DEFIN)
              (SETQ DEFIN (READ INPUT))
              (GO LOOP)))
     EXIT(TERPRI)
          (CLOSEALL NIL])
```

```
(SAVEF
**********************************************************************
  args:       FNAME (Unix filename)
              DEFS  (A LISP list of function names)
              VARS  (A LISP list of variable names)
  calls:      SET, PACK, LIST, MAKEFILE
  uses free:  FFNS, FCOMS
  comments:   A utility function used to write all or a
              portion of the LISP functions, and variables in
              the current LISP environment to a file.  Used
              to create the RPL-INT file.  Also used to
              convert LISP files not created in InterLisp to
              the InterLisp input format.
**********************************************************************
  [LAMBDA (FNAME DEFS VARS)
    [SETQ FCOMS (PACK (LIST FNAME (QUOTE COMS]
    [SETQ FFNS (PACK (LIST FNAME (QUOTE FNS]
    (SET FFNS DEFS)
    (SET FCOMS (LIST (LIST (QUOTE FNS) (QUOTE *) FFNS)
                     (LIST (QUOTE VARS) VARS)))
    (MAKEFILE FNAME])


(SELECT_ALL
**********************************************************************
  args:       TGT (Anything)
              TBL (Untagged relation)
  calls:      MAPCAR, CONS, REVERSE
  called by:  BIF_APPLY, RELATIVE_PRODUCT
  binds:      SET, X
  comments:   Returns an untagged set of all the right
              members associated with the TGT in TBL.
**********************************************************************
  [LAMBDA (TGT TBL)
    (PROG (SET)
          (SETQ SET NIL)
          [MAPCAR TBL (QUOTE (LAMBDA (X)
            (COND
               ((EQ (CAR X) TGT)
                 (SETQ SET (CONS (CDR X) SET]
          (RETURN (CONS (QUOTE Eset) (REVERSE SET])
```

```
(SEQ_TO_REL
**********************************************************************
  args:       S (Untagged LISP list)
  calls:      LEQ, LENGTH, CONS, SEQ_TO_REL
  called by:  EVSEQ, BIF_APPLY, SEQ_TO_REL
  comments:   Returns an untagged relation which is the
              result of converting the RPL input form for a
              sequence to its internal representation.
**********************************************************************
  [LAMBDA (S)
    (COND
      ((LEQ (LENGTH S) 1) NIL)
      (T (CONS (CONS (CAR S) (CADR S))
               (SEQ_TO_REL (CDR S])

(SHOW_ATOM
**********************************************************************
  args:       X (Any LISP atom)
  calls:      ATOM, STRINGP, MEMB, PRIN1, PRINT_LIST, LENGTH,
              CONS, LIST, SPACES
  called by:  DISPLAY, SHOW_ENV, PRINT_LIST
  comments:   An auxiliary function for output of RPL atoms.
**********************************************************************
  [LAMBDA (X)
    (SPACES 1)
    [COND
      ((ATOM X) (PRIN1 X))
      ((STRINGP X) (PRIN1 X))
      [(MEMB (CAR X) (QUOTE (Eset Erel)))
       (COND
         ((EQ (LENGTH X) 1) (PRIN1 (QUOTE empty)))
         ((EQ (CAR X) (QUOTE Eset)) (PRIN1 (QUOTE %()
           (PRINT_LIST (CONS (QUOTE set) (CDR X)))
           (PRIN1 (QUOTE %))))
         (T (PRIN1 (QUOTE %()
            (PRINT_LIST (CONS (QUOTE rel) (CDR X)))
            (PRIN1 (QUOTE %)]
      [(EQ (CAR X) (QUOTE closure))
       (COND
         [(EQ (LENGTH X) 4)
           (PRIN1 (LIST (CAR X) (CADR X) (CADDR X]
         (T (PRIN1 X]
      (T (PRIN1 (QUOTE %())
        (PRINT_LIST X)
        (PRIN1 (QUOTE %)]
    (SPACES 1])
```

```
(SHOW_ENV
**************************************************************
  args:       ENV
  calls:      MEMB, LEQ, WRITE, SHOW_ATOM, PRINT, TERPRI,
              LENGTH, LIST, SHOW_ENV
  called by:  SHOW_ENV
  binds:      X
  uses free:  OPNAMES
  comments:   First implementation for the "env" command.
              Shows the evaluated form of the environment.
              Not currently used, left if wanted for future.
**************************************************************
  [LAMBDA (ENV)
    (PROG (X)
          (SETQ X (CAR ENV))
          (RETURN (COND
                    ((MEMB (CAR X) OPNAMES)
                      (WRITE (QUOTE (System Defined Functions)))
                      (TERPRI))
                    (T [COND
                         ((LEQ (LENGTH X) 4)
                           (SHOW_ATOM X)
                           (TERPRI))
                         (T (COND
                              [(AND (EQ (CADR X)
                                        (QUOTE closure))
                                    (EQ (LENGTH X) 5))
                                (PRINT (LIST (CAR X)
                                             (CADR X)
                                             (CADDR X)
                                             (CADDDR X]
                              (T (SHOW_ATOM X)
                                 (TERPRI]
                  (SHOW_ENV (CDR ENV])


(TF
**************************************************************
  args:       B (LISP boolean)
  called by:  BIF_APPLY, REQUAL
  comments:   Converts LISP booleans to RPL boolean format.
**************************************************************
  [LAMBDA (B)
    (COND
      ((EQ B NIL) (QUOTE false))
      (T (QUOTE true])
```

```
(TYPE
**********************************************************************
  args:       X (Anything)
  calls:      ATOM, STRINGP
  called by:  DISPLAY, EV_SPECIAL_CASES, INFIXOP, PREFIXOP,
              BIF_APPLY, MEM, REQUAL, RPL_REPEAT,
              SUPERSCRIPT, COERCE_TO_REL
  comments:   A utility function used to trap illegal calls
              to the LISP functions, CAR and CDR.  Returns
              the first element if X is a list.
**********************************************************************
  [LAMBDA (X)
    (COND
      ((OR (ATOM X) (STRINGP X)) (QUOTE atom))
      (T (CAR X]

(WRITE
**********************************************************************
  args:       L (LISP list)
  calls:      PRIN2, MAPCAR, SPACES
  called by:  RPL, SET_USER_ENV, DEF_BINDING, ERROR_HANDLER,
              EXIT, FILE_READ, DISPLAY_ENV, READ_USER_DEFS,
              SHOW_ENV
  binds:      X
  comments:   A utility function which alters LISP output to
              a more natural form without parentheses.
**********************************************************************
  [LAMBDA (L)
    (MAPCAR L (QUOTE (LAMBDA (X) (PRIN2 X) (SPACES 1]

(WRITE_USER_DEFS
**********************************************************************
  args:       FNAME (Unix filename)
  calls:      OUTFILE, MAPCAR, CLOSEALL, OUTFILEP, REVERSE,
              PRINT
  called by:  EXIT
  binds:      OUTPUT, DEFOUT, X
  uses free:  USERDEFS
  comments:   A utility function used to write the current
              RPL session's commands to a file, FNAME.
**********************************************************************
  [LAMBDA (FNAME)
    (PROG (OUTPUT DEFOUT)
          (SETQ OUTPUT (OUTFILEP FNAME))
          (OUTFILE OUTPUT)
          (SETQ DEFOUT (REVERSE USERDEFS))
          [MAPCAR DEFOUT (QUOTE (LAMBDA (X)
                                        (PRINT (CDR X)
                                               OUTPUT]
          (PRINT (QUOTE EOF) OUTPUT)
          (CLOSEALL NIL])
```

APPENDIX G - EXAMPLES OF RPL PROGRAMS

A. INTRODUCTION

The purpose of this appendix is to illustrate two example RPL programs which demonstrate the flexibility and potential power of the language, and also some of the design issues involved in the implementation.

B. EXAMPLE #1 - PAYROLL

Suppose there is a file of employee records which is keyed upon a unique employee number. These records contain only the employee name and accumulated number of hours worked for payroll purposes.

In RPL this file can be defined as a simple relation which relates the employee number to the employee record. The employee record is just another relation between field names and their associated values. This file will be refered to as the 'OldMaster' file.

In addition to the 'OldMaster' file an 'Updates' file which would contain only an employee number related to the number of hours for a given time period is required. Again, this file can be represented by a simple relation.

What is desired is a program that will take the 'OldMaster' and 'Updates' files and produce a new updated master with current accumulated hours.

In essence, the values of the 'hours' field in the 'OldMaster' file need to be increased by the amount of hours in the 'Updates' file. A function to do this can be developed from built-in RPL operators and takes advantage of the infix to prefix conversion functional, the functionals which fix one of the two normal infix operator arguments, and several combining functionals. The power of RPL is that this complicated sequential process of many steps can be combined into virtually two steps using RPL constructs. Figure G-1 shows a RPL program that would accomplish the task.

```
F == (file "OldMaster")                                    (1)
U  ==  (file  "Updates")                                   (2)
H == "hours"                                               (3)
sumhrs == ((op +) o ((rsec sel H) !! I))                   (4)
u == ((F # U) rp (as o ((lsec H ,) o sumhrs)))             (5)
F' == ((u # F) rp (op ;))                                  (6)
val F'                                                     (7)
```

NOTES:

```
(1)  F      = old file
(2)  U      = update file
(3)  H      = Field name for hours worked
(4)  sumhrs = Update  auxiliary function to add old hours
              to the update hours
(5)  u      = Updating function
(6)  F'     = New file
(7)  Display file in evaluated form
```

Figure G-1 -- Payroll Example

Notice how the 'op' functional is used to allow infix operators to be combined without any arguments. Likewise, 'lsec' and 'rsec' are used to fix the left or right

argument, respectively. All operators in this example are explained in detail in Appendix C.

F, U, and H are all just data definitions to initialize the names. 'sumhrs' is just an auxiliary function which performs the addition required and also makes the program a little easier to read. The updating function, u, really creates an extensional function in the form of a relation (table) which contains the updated 'hours' field. The new file is created when an ordered union (;) is performed between the records of the update table produced by u and the original file, 'OldMaster'. The ordered union replaces the value of the 'hours' field in the original file with the new value contained in the update table. Normally, the new structure would be saved for use as the 'OldMaster' the next time an update would be scheduled, but the program in Figure G-1 simply displays the resulting file for the user to review.

This example demonstrates the complexity of the language that had to be dealt with in the implementation, but also gives one a feeling for the abstraction, flexibility and power that can be obtained.

C. EXAMPLE #2 - DEVELOPMENT OF 'xi'

The RPL operator 'xi' filters a sequence given a predicate to test its elements. In order to better illustrate the need and execution process of this operator

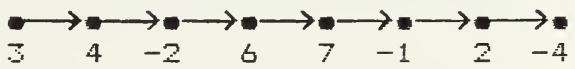
the following RPL sequence will be used:

s == (seq 3 4 -2 6 7 - 1 2 - 4)

This is represented internally as,

(rel (3 4) (4 -2) (-2 6) (6 7) (7 -1) (-1 2) (2 - 4))

and graphically as



Suppose, the user wanted to eliminate the negative nodes in the sequence. The normal filter operation is not suitable since it would simply test both the left and right member of each pair in the relation and eliminate the entire node if either element was negative. The result of performing a normal filter on s would produce:

(rel (3 4) (6 7))

Graphically, this is:

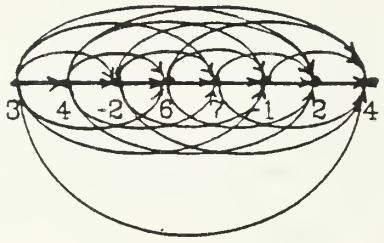
Notice that the resulting elements of the sequence are disconnected and the valid element, '2', has been erroneously deleted. A solution to this problem would have to reconnect the disconnected nodes and not eliminate valid ones by mistake. Thus, the 'xi' operator is justified.

The 'xi' operator accomplishes this process in basically three steps. First, the transitive closure of the sequence is computed. Second, the undesireable nodes are

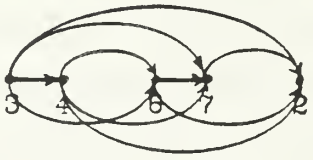
elimininated and third, the redundant edges are eliminated.

This process is illustrated graphically on the sequence s.

(1) Compute (s sup +):

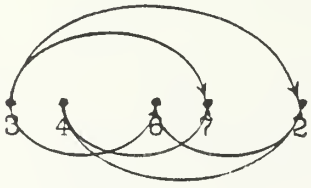


(2) Eliminate negative nodes using restriction,
s restr (rsec > 0):

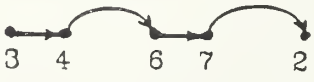


(3) Eliminate redundant edges using mu, a relation minimization operator defined as: (R \ (R ¦ (R sup +)):

s ¦ (s sup +):



s \ (s ¦ (s sup +)):



Thus, the definition for 'xi' follows:

```
p xi r == (mu ((r sup +) restr p)
```

The major implementation problem here is the large amount of temporary storage required to hold the transitive closure of s. The use of LISP as an implementation language eliminated this concern since it already has a built-in storage management system.

LIST OF REFERENCES

1. Backus, J., "Can programming be liberated from the von Neuman style? A Functional Style and its Algebra of Programs", CACM 21, v. 8, pp. 613-641, August 1978.

2. MacLennan, B. J., Naval Postgraduate School Technical Report NPS52-8-012, Relational Programming, September 1983.

3. MacLennan, B. J., Overview of Relational Programming, SIGPLAN Notices, vol. 18, #3, pp. 36-45, March 1983.

4. Futaci, S., Representation Techniques for Relational Languages and the Worst Case Asymtotical Time Complexity Behavior of the Related Algorithms, MS Thesis, Naval Postgraduate School, Monterey, California, June 1982.

5. MacLennan, B. J., Principles of Programming Languages, Holt, Rhinehart, and Winston, 1983.

6. Teitelman, W., Interlisp Reference Manual, Xerox Palo Alto Research Center, 1974.

7. Winston, P. H., and Horn, B. K., LISP, 2d Edition, Addison-Wesley Publishing Company, 1984.

8. Bates, R. and others, ISI-Interlisp Users Guide, 3d Edition, USC Information Institute, 1983.

9. McGilton, H. and Morgan, R., Introducing the UNIX System, McGraw-Hill Book Company, 1983.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Professor Bruce J. MacLennan Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	4
4.	Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, Virginia 22217	1
5.	LCDR Stephen G. Mitton 54 North Walker St. Taunton, Massachusettes 02780	2
6.	CPT John R. Brown 13 Riley Rd. Portville, New York 14770	2
7.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943-5100	1

214072

Thesis
B8144
c.1

Brown
Relational programming: design and implementation of a prototype interpreter.

thesB8144
Relational programming: design and impl
3 2768 000 62589 1
DUDLEY KNOX LIBRARY